Always-Available Static and Dynamic Feedback

Michael Bayne Richard Cook Michael D. Ernst
University of Washington
{mdb,rcook,mernst}@cs.washington.edu

ABSTRACT

Developers who write code in a statically typed language are denied the ability to obtain dynamic feedback by executing their code during periods when it fails the static type checker. They are further confined to the static typing discipline during times in the development process where it does not yield the highest productivity. If they opt instead to use a dynamic language, they forgo the many benefits of static typing, including machine-checked documentation, improved correctness and reliability, tool support (such as for refactoring), and better runtime performance.

We present a novel approach to giving developers the benefits of both static and dynamic typing, throughout the development process, and without the burden of manually separating their program into statically- and dynamically-typed parts. Our approach, which is intended for temporary use during the development process, relaxes the static type system and provides a semantics for many type-incorrect programs. It defers type errors to run time, or suppresses them if they do not affect runtime semantics.

We implemented our approach in a publicly available tool, DuctileJ, for the Java language. In case studies, DuctileJ conferred benefits both during prototyping and during the evolution of existing code.

Categories and Subject Descriptors: D.2 software engineering, D.3.3 language constructs and features
General Terms: Design, Languages, Experimentation
Keywords: dynamic typing, gradual typing, hybrid typing, productivity, prototyping, refactoring, static typing, type error

1. INTRODUCTION

Developers rely on both static and dynamic feedback when creating software. They obtain static feedback, in the form of syntax and type checking, by running the compiler. They obtain dynamic feedback by executing the software and its tests. Only the developer knows what form of feedback is most useful at any given moment during software development, yet the developer is constrained by current tools and cannot always get the feedback they need.

A developer who chooses a dynamically-typed language forgoes the many benefits of static types entirely. A developer who chooses a statically-typed language is denied the ability to obtain dynamic feedback during the periods when their program fails to type-check. Most statically-typed languages embody the philosophy that an ill-typed program is of zero value—the compiler simply rejects it. We consider such programs to have value, in that a developer may be interested in execution paths that do not traverse the type-incorrect code or that are not affected by the inaccuracies in the source code's type annotations.

For what are sometimes technical and sometimes ideological reasons, programmers are denied the benefits of having static and dynamic feedback any time they deem it useful. This state of affairs leads to frustration and wasted effort. We believe that the programmer should be in charge, and should be able to do either form of checking at any time during the development process, and with minimal extra effort on their part [28].

We posit that a developer should write code in a statically-typed language, making a best-faith effort to get the types right and obtaining sound feedback from the type checker. But, even at moments when the program is not globally type-correct, the developer should be able to run the code to test it or to increase insight.

We seek to streamline the process of creating code that will ultimately be type-correct, while recognizing that type-correctness is not always the developer's highest priority. During development, code often temporarily exists in a state of partial type-correctness while the developer focuses on non-type-related aspects of the code. Eventually the code will be made type-correct, but in the order deemed most efficient by the developer. In our system, type annotations are not optional in the source code, so it is unlikely that a user will ignore them or think in a "non-typed" mindset during development. Furthermore, at any moment the developer can obtain complete, sound type-checking feedback by running the standard compiler. Developers can switch seamlessly between the typed and untyped views of their program, since sometimes it is best to fix compiler errors and sometimes it is best to run the program. DuctileJ does not provide all the benefits of a dynamically-typed language: it focuses on the ability to execute the code at any time.

Our goals and approach differ from research that aims to mix static and dynamic types in the same program. The two main approaches are: adding optional static type checking to a dynamically-typed language, or relaxing the type system of a statically-typed language. We briefly describe each in turn (also see the related work in Section 5).

Using a dynamically-typed language with an added static type checker, the developer can obtain dynamic feedback at any time, per usual for a dynamic language, and can obtain static feedback when desired by running the type checker. However, most dynamically-typed languages do not afford the easy addition of a static type system. Dynamically-typed programs tend to explicitly leverage highly-dynamic features like late binding of names, meta-programming, and "monkey patching", the ability to arbitrarily modify the program's AST. Most attempts to layer a static type system atop a dynamic language [3, 19, 34] support only a subset of the language, excluding many dynamic features and compromising the programming model and/or the type-checking guarantee.

The most common approach to relaxing a static type system is to introduce some form of `Dynamic` type (also known as `Object`, `any`, `void*`, etc.), allowing the developer to mix statically-typed and dynamically-typed code in the same program. Such an approach does not meet our goals of providing always-available static and dynamic feedback. Such a program can have static type errors and be rejected by the compiler, thereby preventing the programmer from executing the program. Even if the type checker passes, the program may still fail at runtime because `Dynamic` was used to mask behavior from the type checker: the user did not get comprehensive, effective static feedback. Another disadvantage of this approach is programmer effort: the programmer must explicitly specify which parts of the program are to be statically-typed and which are to be dynamically-typed.

We propose a new approach to relaxing a static type system. Rather than extend an existing language with dynamic elements, we provide an alternative semantics for the language in which the compiler warns about errors in the declared types but always generates code, which performs all type checking at runtime. By providing a semantics that defers static

This research was supported by NSF grant CCF-1016701.

type errors until runtime and masks them whenever possible, we allow the developer to obtain dynamic feedback on part of their program, even if other parts contain type errors.

Our semantics for the Java programming language, which allows for the execution of many type-incorrect programs, adheres to the following principles:

1. Where correct static types exist, use them to provide behavior equivalent to the original semantics. (In Java, static types affect runtime semantics, such as resolution of overloading.)
2. Where correct static types are lacking, use runtime type information to resolve ambiguity in the same manner that static types would be used.
3. Where correct static types are lacking and runtime type information is either incorrect or insufficient to resolve ambiguity, terminate execution with a runtime error.

We implemented this semantics in a tool called DuctileJ. It uses a *detyping transformation* (Section 3) to convert a potentially type-incorrect program into one that is trivially statically type-correct, and which performs all type checking at runtime. Section 4 evaluates DuctileJ: the correctness of its transformation, its usefulness during prototyping and refactoring, and its runtime performance.

In summary, our contributions are:

- A new perspective on delivering the benefits of both static and dynamic typing, which focuses on temporary relaxation of static types.
- An informal semantics and publicly-available tool [14] for the execution of type-incorrect Java programs based on a detyping transformation.
- An evaluation of the benefits conferred by our approach in both prototyping and software evolution scenarios.

2. MOTIVATION

Our proposal is motivated by a simple idea: a developer should always be able to execute their code, regardless of whether the code type-checks. This section describes scenarios where such a capability is useful. Section 4.2 describes case studies that evaluate each of the two scenarios.

2.1 Software Evolution

During the evolution of a software system, global enforcement of static type-correctness can be burdensome to a programmer. It forces changes to be made in an order preferred by the type checker rather than the order preferred by the programmer. Often, the programmer would benefit from the ability to incrementally test changes as they are made—when feedback on changed behavior is most useful—but the programmer is prevented from executing those tests until the entire program is type-correct.

Delaying discovery of a failed test until long after a change has been made forces the programmer to "page in" the code again when investigating the failure, increasing the effort needed to fix the error. Furthermore, if the test failure shows that the change needs to be implemented differently, the developer may have wasted effort performing similar changes in the same inappropriate way elsewhere in the codebase. By testing as they go, the developer can repair errors while the failing code is fresh in their mind, and they can leverage knowledge gained from validating earlier changes to avoid repeating the same mistakes when making later changes.

Representation or interface changes One common evolution task is to change the data representation or interface of a program component. Take as a concrete example an instant messenger (IM) program that initially supports only a single network and is now being expanded to support multiple networks. A user was initially identified by a string: their user name on the single supported IM network. Now the programmer needs to replace all uses of the string type in this context with a record or object type that additionally indicates their network.

Such a transformation—identifying particular uses of a general-purpose type like `String` and converting them to a user-defined type—cannot easily be automated with refactoring tools. The developer must make the change manually, and would prefer to make and test the change incrementally for the reasons stated above. Unfortunately, in a statically-typed language, this is often not possible. One must transform the whole program before type-correctness—and hence testability—is restored.

Even in a well-modularized program, where one might hope to transform and test one module at a time, such changes can impact the interfaces between modules. Once the changes are begun on one module, the other modules are type-incorrect for as long as they reference the old representation. This is true even if they do not depend on the representation, such as a data structure that stores user information. The programmer must either expend effort to specialize their build system to isolate one module at a time for compilation and testing, or concede to the demands of the type system and reinstate total type-correctness before testing.

Exploratory changes When making exploratory changes that include changes to types, a developer will often make the type changes first, preserving the original behavior of the program and stubbing out new functionality. They then go back and make the behavioral changes. This minimizes the period during which they are unable to execute and test their program. However, this also results in a greater time investment in exploratory changes before they are evaluated for viability.

The programmer may discover, shortly after beginning the behavioral changes, that their approach is infeasible. The time spent making the up-front type changes was wasted. With always-available static and dynamic feedback, they can implement and test their type and behavior changes simultaneously. If they determine that they need to take a different approach, they will have done so sooner and with less effort.

Ad-hoc testing Not every aspect of a software system is amenable to automated testing. In some cases developers opt not to use automated testing, even where it is possible, for reasons of expedience. For example, a developer may choose to test their user interface by interacting with it directly. Our approach can benefit these developers by enabling them to perform ad-hoc tests as they make changes. In cases where the type checker forces them to delay this ad-hoc testing until the software is totally type-correct, they may forget exactly which parts of the program's functionality were impacted by their changes and omit ad-hoc tests that are needed.

2.2 Software Creation

Prototyping—creating a rough first draft of functionality—is a low-cost way to evaluate the feasibility of a design or approach. It can reveal hidden assumptions, and it permits rapid iteration and experimentation to hone in on desired behavior.

A certain amount of static typing is beneficial during prototyping. Thinking about types up front, and having them automatically checked, helps the developer to organize the code and the design. This is especially important during prototyping, when a system's foundations are put in place.

But, too much static typing can be detrimental. During prototyping, the structure of the program changes rapidly and code may be quickly written and thrown away. Expending the extra effort to add type annotations, or to create abstractions with separate interfaces and implementations, can be wasteful given the transient nature of the code. Because of this, dynamically-typed languages are considered to be very effective for prototyping.

Prototypes often contain temporary or partially-implemented functionality that falls out of date with the rest of the program. If the code is statically-typed, the programmer has two undesirable and wasteful options. One option is to expend effort to keep the unused (and possibly irrelevant) code type-consistent with the rest of the program. Another option is to comment out the code, which forces the developer to chase down any other bits of code that reference the commented code, and painstakingly identify the precise boundary between what code is used and what code must be commented out.

In a language that distinguishes between static and dynamic constructs (such as the `Dynamic` type), the static parts of the code may fall out of date as the prototype rapidly evolves. These parts of the code then cause the program to fail to compile and require the developer to fix them before they can once again execute their code, just as they would in a completely statically-typed language.

Furthermore, any use of the `Dynamic` type requires the programmer to later revisit the code, perhaps having forgotten their precise intentions in the meantime. Recalling those intentions, particularly in the absence of the helpful documentation of type declarations, then manually updating the types, is a time-consuming process.

A developer should be able to write down the types they think are appropriate, but should not be required to maintain perfect type-correctness while they are prototyping. By having both static and dynamic feedback available at all times, the developer can focus on type-correctness or behavioral-correctness as they see fit, and not be forced to prioritize one over the other. By using the types they think are appropriate, they express their intentions concretely in the code and reduce the effort needed to later make the code type-correct.

Prototypes eventually give way to production code, where static typing can help greatly with code evolution and maintenance. It is possible for developers to create a prototype in a dynamically-typed language, and then rewrite everything in a statically-typed language once the exploratory prototyping phase is complete. Fred Brooks once said to "plan to throw one away; you will, anyhow" [11]. But he now acknowledges that a better practice is to "build a minimal thing—get it out in the field and start getting feedback" [35]. It is quite common for prototypes to evolve into production systems. In such circumstances, there is no good point at which to rewrite the entire project in a new language. The language chosen for the prototype is often the one that is used for the final product. It should be one that supports the dual benefits of dynamic and static typing during prototyping.

3. IMPLEMENTATION

We have implemented our approach for the Java language in a tool called DuctileJ [14]. DuctileJ consists of a plugin to the Java compiler and a run-time library. The compiler plugin performs a *detyping* transformation that converts a normal Java program to one in which type checking is deferred until runtime. The plugin operates on the abstract syntax tree (AST) during compilation, before generation of Java bytecodes. The runtime library uses Java's runtime type information and its reflection mechanism for dynamically accessing object fields and invoking methods.

For nearly all well-typed programs, the detyping transformation yields code with semantically-equivalent runtime behavior (specific deviations are described in Section 3.8). For type-incorrect programs, execution will terminate with a runtime type error roughly only when the program executes a statement that performs a dynamic cast to an incompatible value, calls a method not supported by the receiver, accesses a non-existent field, or performs a built-in operation (like arithmetic multiplication) on a value of invalid type. Further details on execution-terminating runtime errors are provided in the detailed explanation of the transform that follows.

An inconsistency between a declared type and a runtime value does not terminate the program, because the runtime value might provide all the methods and fields that are necessary for this particular execution. If the execution completes normally, then the type inconsistency was not important. (Recall that the programmer only runs the code at moments when they have decided that observing a test execution is more valuable than reviewing type checker warnings.) If the execution fails, then the type inconsistency may yield insight into the failure. Therefore, we propose that the declared types be checked at each (pseudo-)assignment, and any inconsistencies be logged. If the program fails with a runtime type error, then it is reported along with relevant declared type inconsistencies. These could be determined, for example, via dynamic slicing from the runtime type error. This approach contrasts with other approaches that halt the program as soon as a declared type is inconsistent with a runtime value. The error logging and slicing are not yet supported in our DuctileJ implementation.

Space limitations prevent us from providing complete details of the transformation in this paper, so we describe the basic transformation and a few language constructs, and summarize the remainder in Section 3.6. Space limitations also prevent us from describing the prototype implementations of the Ductile approach for C# (`http://code.google.com/p/ductilesharp/`) and for Scala (`http://www.cs.washington.edu/education/courses/cse501/10au/ductilescala.pdf`)— languages that present a different set of design challenges.

3.1 Basic Transformation

Variable Declarations The transformation replaces all declared types with `Object`. This includes local variables, method parameters, and class fields. An example of such a transformation follows:

```
class Foo {
    int bar;
    int compute (int baz) {
        String qux = ...;
    }
}
    ⇓
class Foo {
    Object bar;
    Object compute (Object baz) {
        Object qux = ...;
    }
}
```

In certain circumstances, variable declarations are not transformed. The exception declaration in a `catch` clause is not transformed. The declared type of the exception defines the runtime semantics of the program and thus must be preserved. However, `catch` clauses that reference an unknown type—which would result in compilation failure if preserved in detyped code—are removed as they cannot impact the behavior of the program.

Variables that are declared as `final` and are initialized by a constant expression are not transformed. Such constant declarations must remain typed to preserve correct use in annotations, and in situations where constant boolean expressions impact definite assignment analysis.

Method Invocation and Field Access Object construction, method invocation, and field reads and updates are transformed into calls to the DuctileJ runtime library, which performs these actions via reflection. Such calls in the example code take the form `RT.`*`action`*.

The behavior of the `RT` calls is generally a runtime equivalent of the pre-transformed static code, along with more informative error handling than that provided by Java's reflection library.

The following example shows object construction, field read and update, and method invocation:

```
Foo foo = new Foo();
foo.bar = 3;
int r = foo.compute(foo.bar)
    ⇓
Object foo = RT.newInstance("Foo");
RT.assign(foo, "bar", 3);
Object r = RT.invoke("compute", foo, RT.select(foo, "bar"));
```

The `RT.newInstance` call raises an error at runtime if the supplied class does not exist or lacks a public constructor that accepts the appropriate number of arguments. Similarly, `RT.invoke` raises an error if the receiver lacks a method of the specified name and arity. `RT.select` and `RT.assign` raise an error if the supplied object lacks a field of the specified name.

Note that the types of the arguments to constructors and methods and the types of the values stored in object fields need not be compatible with the declared types in the original code. While method invocation and field access may raise *message not understood* errors, runtime *type* errors are generally only raised when a value of incorrect type is used as an operand of an arithmetic or boolean operator, or in language constructs like `if` or `while` statements, which require a specific primitive type. Section 3.4 discusses the interaction between detyped and undetyped code and describes how method invocation and field assignment *can* result in runtime type errors in those cases.

Operators Built-in unary and binary operators are transformed into calls into the runtime library that check the types of the operator arguments and then perform the appropriate computation. Numeric promotion is performed by `RT.binop` at runtime, per the rules described in Java Language Specification §5.6. If either operand is of incompatible type, a runtime exception will be raised.

3.2 Exception Wrapping

When method invocations are converted into reflective calls, information about checked exceptions is lost. The following transformation results in invalid Java code:

```
int readChar (InputStream in) {
    try { return in.read(); }
    catch (IOException e) { return -1; }
}
    ⇓
Object readChar (Object in) {
    try { return RT.invoke("read", in); }
    catch (IOException e) { return -1; }
}
```

The call to `in.read()` is declared to throw an `IOException` but the call to `RT.invoke` is declared to throw no checked exceptions. The Java compiler rejects any `catch` clause for a checked exception that cannot be thrown in the enclosed code.

Furthermore, when invoking a method via reflection, Java wraps any exceptions thrown by the invoked method in `InvocationTarget-Exception` instances. Thus, even if the compiler did not reject the transformed code above, it would have a semantics that differs from the original code: the `IOException` would not be correctly caught, if thrown.

To ensure both a correct semantics and legal Java code, the runtime library catches and re-wraps any exceptions thrown as a result of its reflective calls into the new exception type `WrappedException`. This new exception type is used in lieu of `InvocationTargetException` to avoid conflicting with direct uses of reflection by the untransformed code. All `try`/`catch` blocks are then transformed as follows:

```
Object readChar (Object in) {
  try {
    try { return RT.invoke("read", in); }
    catch (WrappedException e$W) {
      if (e$W.getCause() instanceof IOException)
        throw (IOException)e$W.getCause();
      else throw e$W;
    }
  } catch (IOException e) { return -1; }
}
```

This ensures that checked (and unchecked) exceptions are caught in the same manner as the untransformed code, and that the Java compiler sees the static possibility that the appropriate checked exception is thrown inside a given `try`/`catch` block.

Note that this transformation is done regardless of whether the `try`/`catch` block originally contained code that threw the exception in question. We see this as another beneficial relaxation of the static type system. A programmer can remove a checked exception from the `throws` clause of a method and defer the additional effort of removing any `try`/`catch` blocks that would normally become invalid as a result of that method no longer throwing the checked exception.

3.3 Method Overloading

Java allows multiple methods with the same name and arity to be declared as long as they differ in their argument types. At compilation time, the static types of the arguments are used to select the appropriate overloaded method. As the detyping transformation converts the types of all method arguments to `Object`, this can result in collisions in overloaded method declarations. In addition to the need to prevent these collisions, transformed well-typed code must make the same choice in resolving overloaded methods dynamically at run time as would have been made statically at compile time.

Name mangling cannot be used in this circumstance, as constructors cannot be name-mangled. Instead we use *signature mangling*. A method's arguments are replicated into value-carrying arguments and type-carrying arguments. The value-carrying arguments have type `Object` and are used at runtime to pass values to the method. The type-carrying arguments have the original declared types and their values are not used at runtime. This ensures that every method has a unique non-colliding signature. An example of signature mangling is:

```
class Printer {
    void print (String value) {...}
    void print (int value) {...}
}
Printer p = new Printer();
int val = 3;
p.print(val);
    ⇓
class Printer {
    void print (Object value, String value$T) {...}
    void print (Object value, int value$T) {...}
}
Object p = new Printer();
Object val = 3;
// 2nd argument is declared type of argument expressions
RT.invoke("print", new Class<?>[]{int.class}, p, val);
```

During the transformation of call sites, overloaded methods are resolved based on the declared types of their arguments. If sufficient compile-time type information exists to unambiguously select a most-specific overloaded method, the argument types identifying that method are injected into the program to communicate to the runtime which method to execute. If no most-specific overload can be determined, no type information is injected. In that case, `RT.invoke` will attempt to resolve the overload using the runtime type of the method arguments. If that type information is also insufficient to resolve the overload, a runtime error is raised.

During runtime method invocation, `RT.invoke` inserts default values (`null`, `boolean`, or the appropriate `0` value) into the argument array at the type-carrying positions.

3.4 Library Code

DuctileJ's transformation differentiates between code that is being transformed and code that remains statically-typed. The user can make this choice; by default, DuctileJ detypes all source code available to it, but not binaries such as platform code and third-party libraries. We refer to untransformed code as *library code*. While a bytecode transformation approach could allow third-party libraries to be detyped even without access to their source code, we expect detyping to be most useful on code that the developer is actively changing. Besides, only type-correct code can be compiled into bytecodes, and DuctileJ is intended to help programmers to run type-incorrect code.

Detyped code need not use libraries in a type-checkable manner. At run time, when detyped code makes a call into library code, if the arguments do not have the parameter types required by the library call, a type error is raised. This is analogous to DuctileJ's handling of primitive operations.

Type signatures are preserved for any method in detyped code that overrides/implements a method defined in a library class/interface. This ensures that if a reference to a detyped instance is passed into library code, the untransformed library code will operate correctly. An example of a detyped library overrider is shown below:

```
class Person implements Comparable<Person> {
    public int compareTo (Person p1, Person p2) {
        int rv = ...;
        return rv;
    }
}
    ⇓
class Person implements Comparable<Person> {
    // type signature unchanged
    public int compareTo (Person p1$P, Person p2$P) {
        Object p1 = p1$P, p2 = p2$P;
        Object rv = ...;
        return RT.cast(int.class, rv);
    }
}
```

Detyped local variables use the same names as the original parameters. This allows the detyped method bodies to use the arguments as l-values in assignments whose r-values are themselves detyped. Also, the return value of the method is dynamically cast back to the declared return type of the method. If the body of the method computes a return value of the wrong type, a type error will be raised by `RT.cast`.

This signature preservation reduces, in a small way, the flexibility of the detyped code. Normal detyped methods can accept any argument type and return a value of any type, regardless of their declared types. Only if the bodies of those methods make use of the values in a manner incompatible with their runtime types will an error be raised. Library overriders must be called with type-correct arguments and return type-correct values, regardless of whether those arguments are used in the method body or the return value is used by the caller.

3.5 Type Resolution

Part of the detyping transformation is purely syntactic, and part requires type resolution for names and expressions, such as when disambiguating overloaded methods (Section 3.3). Even in non-overloaded method invocations, the detyping transformation must distinguish between static and non-static receivers to correctly transform the code.

DuctileJ implements a relaxed type resolver that performs normal type resolution using the types declared in the code, and falls back to `Object` for expressions that cannot be resolved. For a type-correct expression, the resolver obtains the same type as the standard Java compiler, and for type-incorrect expressions, the resolver can proceed in cases where the Java compiler would terminate with a type error, by substituting `Object` for the types of unresolvable sub-expressions.

3.6 Other Transformations

The Java language has grown large in its maturity, and achieving a correct semantics for detyped code required substantial effort. Some other interesting challenges handled by DuctileJ's detyping transformation [14] include: handling of arrays, `final` fields and variables, arithmetic and logical operators, control flow constructs, widening and narrowing conversions, annotation and enum classes, anonymous inner classes, definite assignment analysis, variable-arity methods, partially implemented interfaces, super and chained constructor calls, use of explicit outer `this` pointers in inner class construction, security manager restrictions, and primitive vs. object equality.

3.7 Debugging

The detyped code that is executed differs from the source code that the user views in their IDE. This did not decrease productivity when debugging in our case studies. The stack trace is identical, even down to line numbers, except that between every pair of stack frames is a new stack frame for the DuctileJ runtime library's `RT.invoke` call. Users should ignore those frames. All debugger features such as breakpoints and watched variables work. The changed variable types and the extra method arguments are visible.

Software	SLOC	CLOC	Tests	CLOT
Google Collections (1.0)	50,744	18	44,760	45
HSQLDB (1.8.1.2)	76,378	2	3,783	0
Joda-Time (2.0)	78,466	3	3,736	4

Figure 1: Correctness tests. SLOC is non-comment lines of code. CLOC is changed lines of non-test code. Tests is the number reported by the test framework (e.g., JUnit), plus the number of SQL commands (for HSQLDB). CLOT is changed lines of test code.

3.8 Limitations

This section describes two Java language features that DuctileJ does not yet support, and our plans for supporting them. DuctileJ does not currently support separate compilation, but we do not have space to explain our design.

Reflection The detyping transformation changes both the arity and type signature of methods (including constructors). DuctileJ does not yet support code that uses reflection to inspect or call methods that take arguments.

One approach to supporting reflection is to intercept calls to Java's reflection routines and translate them such that they report metadata that matches the original undetyped code. DuctileJ would also transparently insert appropriate values for type-carrying arguments to method and constructor calls made via reflection. This can be done by transparently rewriting bytecode as it is loaded into the VM, which will ensure that reflection use in both detyped and library code functions properly. The feasibility of such an approach has been established by the EMPL [42] and RuggedJ [26] tools, which hide program transformations from Java reflection.

Serialization Java provides built-in support for converting objects to and from a binary representation for network transmission or persistent storage. Like reflection, this facility relies on metadata that is modified by the detyping transformation. The Java serialization framework provides hooks for custom object serialization, which can be used to automatically generate custom serialization and deserialization routines that read and write the standard serialized representation of the untransformed class.

4. EVALUATION

The goals of our evaluation are threefold. First, we wish to demonstrate that our tool, DuctileJ, correctly performs the detyping transformation on a sufficiently large subset of the Java language in order to render it usable on real-world programs. Second, we wish to understand the benefits of always-available static and dynamic feedback during prototyping and software evolution tasks. Third, we characterize the performance of DuctileJ.

4.1 Semantic Correctness

Much of the challenge in implementing the detyping transformation in Java stems from handling the complicated interplay of language features like inner classes, variable-arity methods, generics, and autoboxing. Because such features, not to mention their interactions, are generally omitted from formalizations of the Java language, we chose an informal approach to demonstrating the semantic equivalence of detyped to undetyped code.

We used DuctileJ to compile and run the unit test suites of a variety of Java libraries and applications and confirmed that the suites reported no failures. While this does not establish that there are no differences between the detyped and standard Java semantics—indeed, Section 3.8 lists some—it does build confidence that there are no commonly used language features, or interactions thereof, that are not handled correctly by the detyping transformation.

Figure 1 summarizes the largest of the projects that we used in our evaluation. All tests passed when run in detyped form. We made minor modifications to work around the limitations described in Section 3.8.

Reflection The Google Collections test suite uses reflection to succinctly express tests parameterized over all methods of a class, and over individual parameters to those methods. We modified the test drivers to ignore the type-carrying parameter positions and to obtain the types of the value-carrying positions from the type-carrying positions. These modifications did not change the conditions for correctness. The Joda-Time test suite uses reflection to invoke the constructors of two classes which do not provide public constructors. HSQLDB uses reflection to fall back to a simpler database metadata class if the construction of the newer class results in failure. We manually inserted type-carrying arguments into these calls.

Section 3.8 describes an approach to handling reflection in detyped code, which would eliminate the need for these changes.

Serialization The detyping transformation inserts initializers for detyped primitive fields to emulate the initialization of said fields to their default values (e.g., `0` for an `int` field). These injected initializers are run as a part of an object's constructor, but an object's constructor is not invoked during deserialization, which circumvents normal constructor invocation. We manually inserted, into Google Collections, 3 custom `readObject` methods that initialize detyped primitive fields to their default values before triggering the normal deserialization process. DuctileJ should synthesize this `readObject` method automatically during the detyping process, but this is not yet implemented.

In the Joda-Time test suite, 48 tests consisted solely of deserializing objects from binary data stored with the project and validating the resulting objects. Because detyping changes the serialized form of the objects in question, we used detyped instances to regenerate the binary data against which the tests were run. DuctileJ should synthesize `readObject` and `writeObject` methods to overcome this incompatibility, but this is not yet implemented.

4.2 Case Studies

To evaluate the benefits of always-available static and dynamic feedback, we used DuctileJ to prototype new programs as well as to evolve existing, type-correct programs. We were specifically interested in learning which of the many benefits conferred by the use of dynamically-typed languages could be obtained by using DuctileJ. We also wished to investigate the benefits of fluidly switching between static and dynamic views of the same code.

The developers were two of this paper's authors. One has developed commercial software for 14 years, chiefly in statically-typed languages like Java and C++, though doing many smaller projects in scripting languages (Perl, PHP, Ruby, and Python). The other is a senior engineer at Microsoft with 11 years of professional software engineering experience, of which one year was using dynamically typed languages such as Ruby and Python. Based on their experiences, both developers have a preference for static typing. Thus, they began with skepticism about DuctileJ, but strove to maintain an unbiased and objective viewpoint.

These initial studies are limited, so the conclusions might not generalize to other developers or tasks; future work should extend them.

4.2.1 Prototyping

Two developers each implemented a contact management application. It was specified as follows: "Create a simple program for managing an e-mail address book. Each entry in the address book has fields for first and last name, e-mail address, and birthday. The software must support entry creation, editing, browsing, and lookup, via a GUI or a command-line tool. Structure the program in two parts: a database that stores address book entries with separate interface and implementation, and the application built atop that library." The two completed applications were 668 and 702 lines long (including test code) and took 4 and 5.5 hours to write, respectively.

We next describe the benefits of the Ductile approach that the developers leveraged. They worked independently, but their experiences were similar, so the following text reports the experience of the first developer described above. In each case, statements reflect the developer's self-reported beliefs while performing the case study.

Duck typing Duck typing checks each individual field access or method invocation at runtime and succeeds or fails based on whether the required field or method is defined for the object. It is akin to structural typing where structural conformance is checked at runtime.

In the prototype, the interface portion of the application made use of a `Database` interface, which evolved as the prototype was developed. The developer used the `Database` type name in code that he knew would operate on the database abstraction, but he did not explicitly declare which methods would be exported by the `Database` interface. Instead, he developed a concrete implementation of the interface and deferred decisions about which methods in the concrete implementation would be exported via the interface and which would not. Once the prototype was complete, he examined the type checking output, which had been available all along but which he had been temporarily disregarding. He immediately saw which implementation methods were needed in the interface; all other methods could remain hidden as implementation details. This saved him the effort of keeping interface declarations up to date with the appropriate implementation definitions as the prototype evolved.

Checked exceptions Java distinguishes between checked and unchecked exceptions. For calls that might generate a checked exception, either the call must occur in a `try`/`catch` block that handles the exception, or the enclosing method must declare that it allows the exception to propagate. The developer was able to defer the handling of checked exceptions until his design had reached a point of stability.

This conferred multiple benefits. The main benefit was that, as the design took shape, he was able to focus on whether the code effectively provided the desired functionality, rather than be distracted by the minutiae of failure handling. Some aspects of failure handling are important, but if a potential design is deemed to be inappropriate, effort put into handling specific error cases while prototyping that design is most likely wasted. The developer reported that he was able to more cheaply experiment with alternative designs, and only invest the effort in proper error handling once he had reached a final design.

This saved the developer from the temptation of inserting placeholder error handling code which he might have easily overlooked when later finalizing the design. He did not want to insert dummy `try`/`catch` blocks just to get his prototype working, because he would have had to later manually find and repair all such temporary code without the help of the compiler. Further, he feared that he may have placed his temporary error handling at places that, after finalizing his design, he decided were not appropriate. This would have added to his workload, to remove the temporary error-handling code from those locations, add the necessary `throws` clauses, and then add the real error-handling code in the desired locations.

Finally, by delaying the declaration of the checked exceptions propagated by a method, he was confident that he avoided polluting his method signatures with vestigial exception declarations. Given the inability to defer checked exception handling, developers often choose to propagate checked exceptions while prototyping to avoid having to decide how and where to handle errors. These exceptions then end up in method signatures. The developer may subsequently change the implementation such that it throws a different set of checked exceptions. The compiler, unfortunately, does not report that he now has unnecessary checked exception declarations in his methods since it is perfectly legal in Java to declare that a method may throw checked exceptions that it does not actually throw. Thus, these checked exceptions often linger in the method declarations, complicating client code which must now handle these phantom error conditions. By delaying the declaration and handling of checked exceptions until the end of the prototyping process, the developer reduced the likelihood of such signature pollution.

Access control An unexpected benefit the developer encountered was the ability to defer the enforcement of access control. In converting his prototypes to type-correct code, the developer discovered that he had unintentionally called private or protected methods from code that did not have access to those methods. He was glad that he was not using the stock Java compiler, which he felt would have distracted him from his prototyping task to determine the correct way to accomplish the desired functionality.

Honoring access control restrictions is clearly something that should be done eventually, but he felt it was beneficial to defer that effort until after he knew that the code in question would make it into the final design. Earlier versions of his code (that he later discarded) called other inaccessible methods. He felt that, in such cases, he was able to avoid wasting effort determining how to properly access functionality that was eventually not needed.

Negatives The developer encountered some annoyances while performing the case study, but none of them delayed him more than a moment. One was that he once mistyped a field name, and he discovered this fact via testing rather than at compile time, when he would have preferred. The other negatives have to do with limitations of our prototype implementation. The developer tried to use reflection to obtain the return type of a method, but it returned `Object` instead (see Section 3.8). The developer was happy not to have to write import statements, but the import statements were necessary to resolve certain constructor calls, which was a surprise when he had been using the type in other contexts in the same file. Finally, because he was using an early version of DuctileJ, he encountered some bugs in it, which have since been fixed; interrupting his thought process to report and work around these was his biggest annoyance in the study.

Comparison with automated refactoring Modern IDEs support certain type-related code changes. To call these "automatic fixes", however, would be a misnomer. In our study, the developer recognized that changing access modifiers, adding thrown exceptions, adding methods to interfaces, etc.—just to satisfy the compiler at an intermediate point in the program's evolution—would degrade his design. It would require human effort in the future to find places that needed to be corrected, and to correct them. The Eclipse "fix" would make a developer more likely to introduce undesired dependences that would linger unnoticed and be difficult to fix later; with the Ductile approach, the regular compiler can still find them. Eclipse's refactorings are useful in certain circumstances, but in this case Eclipse would have automated the code transformations that the developer had already decided were ill-advised.

4.2.2 Software Evolution: Evaluating a Refactoring

We hypothesize that our approach confers benefit for software evolution tasks by allowing design alternatives to be explored with less effort. To investigate this hypothesis, the first developer described in Section 4.2.1 refactored JHotDraw (`http://www.jhotdraw.org/`), a GUI framework for technical and structured graphics.

JHotDraw defines a `Figure` interface, which serves as the root of a rich hierarchy of classes that model geometric figures. `Figure` defines a method `containsPoint(int x, int y)`, which the developer wished to refactor to `containsPoint(Point p)`.

The developer's goal was to expend as little effort as possible evaluating whether this refactoring was feasible and appropriate. One way to make that evaluation is to perform a "vertical slice" of the refactoring: change the interface, any necessary implementations, and one or more test cases that invoke the refactored code. The developer selected one concrete implementation `TriangleFigure` and its associated test case `TriangleFigureTest`. Thus, the goal was to get `TriangleFigureTest` running against the refactored interface. Because he may determine at that time that the refactoring is inappropriate, he would like to expend as little effort as possible in doing so. After evaluating the refactoring, the developer would either undo the changes or complete the refactoring.

The heart of the refactoring is three key changes: change the `containsPoint` method signature in the `Figure` interface; update the `TriangleFigure.containsPoint` implementation accordingly; and change the `TriangleFigureTest` code to supply `Point` arguments instead of *x*- and *y*-coordinates.

The developer investigated three approaches. A type-driven manual refactoring required 24 code changes to make the test case pass. A type-driven refactoring with Eclipse support required one Eclipse refactoring, then 16 manual edits (a few of them tricky, as described below) to make the test case pass. A manual refactoring using DuctileJ required only the 3 key changes before the test case passed.

We now discuss each of the approaches in turn.

Type-driven manual refactoring The type checker can guide a developer through a refactoring. The programmer performed the three key changes, then ran the compiler, which reported 30 errors.

Of the 30 errors, 9 stemmed from calls to `containsPoint` that had not yet been updated to supply a `Point` instead of separate *x*- and *y*-coordinates. The other 21 errors stemmed from concrete implementations of `Figure` that lacked an implementation of the new `containsPoint(Point)`, as their `containsPoint(int,int)` method had not yet been converted.

The programmer performed minimal fixes for the 30 remaining errors so that he could compile and execute `TriangleFigureTest`. This required only 21 changes to the source, because updating the implementation of `containsPoint` in some abstract base classes resolved errors for all of their children. After making those changes, he was able to execute `TriangleFigureTest` and evaluate his refactoring. Those 21 changes represent substantial effort beyond the 3 changes that were directly related to his evaluation.

Eclipse-supported semi-automated refactoring Eclipse provides a *Change Method Signature* automated refactoring to assist with refactorings like the one in this case study. The developer instructed Eclipse to change the signature of the `Figure.containsPoint` method by removing the `x` and `y` parameters, adding a `java.awt.Point p` parameter, and using `new java.awt.Point(x, y)` as the default argument for calls. He would have preferred a default of `new Point(x, y)`—and, in fact, tried that first—but Eclipse fails to insert necessary `import` statements. Though the final Eclipse refactoring littered the code with undesirable fully-qualified class names, which must be manually fixed later, it allowed him to defer that effort until after he had evaluated his refactoring by executing his desired tests.

After automated refactoring, the compiler reported 34 errors. 16 of these errors stemmed from implementations of `containsPoint` that were referencing the old formal parameters `x` and `y` rather than the new parameter `p`. The other 18 errors resulted from Eclipse's verbatim insertion of `new java.awt.Point(x, y)` as the argument to existing calls to `containsPoint`. Eclipse made transformations like the following:

```
public boolean test (Figure fig, int x1, int y2) {
    return fig.containsPoint(x1, y1);
}
     ⇓
public boolean test (Figure fig, int x1, int y2) {
    return fig.containsPoint(new Point(x, y));
}
```

The refactoring ought to substitute the previous actual arguments for `x` and `y` in the default expression. Only a few calls to `containsPoint` happened to use precisely the arguments `x` and `y`, and were correspondingly correct after the naive verbatim text replacement.

Manually repairing the resulting errors required 16 changes to source files. Most of those changes were straightforward, but two types of changes presented more difficulty. First, because the automated refactoring erased the old parameters to the `containsPoint` call, there were several situations where he had to make use of the version control system to inspect the prior arguments to determine the correct values to use in constructing the new `Point`. Second, in 3 cases, the original arguments were *not* the variables `x` and `y`, yet *other* variables named `x` and `y`, with correct types, happened to be in scope at the call. The verbatim default argument insertion thus resulted in code that compiled but whose behavior was inconsistent with the behavior prior to the refactoring.

DuctileJ-supported manual refactoring When using DuctileJ, the developer made the desired change to `Figure.containsPoint` and

Software	Stock (s)	Detyped (s)	Slowdown (×)
Google Collections	42.8	286.0	6.7
HSQLDB	15.0	16.3	1.1
Joda-Time	8.5	65.9	7.8

Figure 2: Average execution time of test suites.

updated the code in `TriangleFigureTest`. After just these 2 changes, he was able execute the refactored code. DuctileJ reported that a runtime type error was encountered in the `TriangleFigure.containsPoint` implementation. He then converted that method to the new signature and updated its body. At this point, he executed the unit test successfully and was able to evaluate his refactoring.

DuctileJ allowed him to execute the desired tests without making any changes unrelated to his immediate goal. It also guided him toward exactly the changes that were needed to "bridge the gap" between the updated `Figure.containsPoint` method and the updated `Triangle-FigureTest` class, which made calls using the new signature.

While this case study is simple, and could be better supported by improved refactoring tools in the Eclipse IDE, developers frequently perform refactorings that are even more complex and must be done with limited or no automated refactoring support [27]. It is impossible to predict every kind of refactoring a developer may wish to perform and support it with special-purpose machinery in the IDE. However, the Ductile approach can simplify the process of evaluating any refactoring, increasing confidence that it is appropriate before the developer commits to the potentially substantial effort of applying that refactoring to the entire codebase.

4.3 Performance

The detyping transformation degrades execution performance. Our focus so far has been to establish the feasibility and utility of our approach, without undue concern for performance. In the case studies, the developers observed no slowdown, perhaps because: GUI applications are not compute-bound; untransformed libraries often dominate execution time; and test cases tend to execute quickly. Figure 2 summarizes the slowdown exhibited on the unit test suites we used to evaluate semantic correctness in Section 4.1. We have found the performance to be acceptable for a development-time tool.

There are two major sources of slowdown in detyped code. The first is the dynamic lookup and reflective invocation of methods, and the second is the dynamic execution of arithmetic and logical operations.

Dynamic method dispatch Dynamic method dispatch incurs performance penalties in two ways: the correct method to be called must be resolved, and the method must then be invoked via Java's reflection mechanism. DuctileJ caches method resolution results.

Performance for calls with dynamically correct type information could be improved by inserting a type check on the receiver, then performing a normal inline method call if the type check succeeds.

Performance for calls without correct type information could be improved by generating and loading shim classes, then invoking them through a standard `invokeinterface` bytecode instruction, as in JRuby [23]. An even more appealing approach uses the forthcoming `invoke-dynamic` support in the Java 7 VM [33].

Arithmetic and logical operations Normal compiled Java code uses special bytecode instructions for operations on primitives. In detyped code, DuctileJ incurs the overheads from boxing, unboxing, dynamic type tests, promotions, and conversions.

One approach to improving the performance of such operations is to avoid detyping variables declared with primitive types. This would improve runtime performance at the expense of reducing the flexibility of detyped code—it would no longer be possible to carry arbitrary values in variables declared with primitive types.

The approach could be taken further by not applying the detyping transformation to any code that type-checks. However, if DuctileJ did so, then a type-check and a cast would be required whenever calling such code, just as with undetyped libraries (Section 3.4). It would no longer be possible to run code in which a client passes an object that contains all the methods that will ever be called at run time, and this is a major feature of DuctileJ. Our goal in this research was to maximize development-time flexibility, but it is possible that greater productivity may be achieved via different performance/flexibility tradeoffs in different development scenarios. Future work should investigate these tradeoffs.

5. RELATED WORK AND DISCUSSION

Our work aims to integrate dynamic and static type systems in a single language. Previous work on this topic generally falls into two categories: strengthening a dynamic type system without losing the feel or expressiveness of the dynamic language, and adding a `Dynamic` type to a statically-typed language while retaining as many guarantees as possible about the statically-typed portion of a program. There is significant variation within the two categories, and also some notable outliers. Our approach differs from all of this previous work, both in motivation and in technical details. This section provides an overview of related work and then highlights differences.

5.1 Strengthening a Dynamic Type System

Adding types to a dynamically-typed language can improve performance and detect some errors at compile time. In order to retain the feel of the language, programmers are not required to write type annotations; instead, type inference is used. This general approach is often called soft typing [13]. First applied to Scheme, it has since been extended to other languages [29, 25, 4, 12, 19].

Soft typing never rejects a program, but it does issue warnings. The program can still fail due to a type mismatch at runtime. Such errors do not corrupt the underlying runtime system, because a dynamically-typed program performs checks before each operation that requires a given type, such as primitive operations or field accesses. In addition to issuing warnings, soft typing also improves efficiency by optimizing away unnecessary runtime checks, when the type inference can statically prove that the check is guaranteed to succeed at runtime.

This approach has also been applied in a more practical setting, in the RPython (for "Reduced Python") language [3]. RPython eliminates enough of the dynamism of the full Python language that type inference always succeeds, if the program has no errors. Its goal is to achieve higher performance, enabled by type guarantees. DRuby [19] follows this approach, but is focused on discovering type errors in existing Ruby programs. The PRuby [19] extension gathers dynamic profiles in order to reduce programmer annotation burden, and it also replaces untypable dynamic features with statically-analyzable alternatives.

Researchers report that many programs written in dynamic languages are nearly typable, because they don't use much dynamism in practice [6, 34, 18, 22, 32]. However, practitioners have not adopted these restricted dialects or type-inference mechanisms.

In practice, it is difficult to increase the amount of static typing in any language. The largest-scale conversion was probably the addition of `const` to C/C++ programs, an arduous process commonly referred to as "`const`-correctness hell". A close second is addition of generic types [10] to Java programs. Years after the introduction of generics, Sun/Oracle has not yet converted all of the signatures in the JDK to use generic types, and they never intend to convert the bodies to be generics-correct. Lisp programs have been retrofitted with `declare` and `proclaim` for efficiency, and the authors of Typed Scheme [44, 45] have rewritten part of their systems to use it. In each of these examples, the conversion required hard thinking and program restructuring, though the conversion generally did improve the code.

Our experience is that when programmers start out with a dynamic mindset, they may choose a design that is not statically-typable, even when a statically-typable design exists and is preferable. This leads to painful program restructuring much later in the development process. Thinking about and writing the types helps to prevent this problem, and being able to run a type checker helps even more. This is why we believe that types should be in the programmer's mind and code, and be supported by the toolset (but not enforced until the programmer is ready), from the beginning of the development process.

5.2 Adding `Dynamic` to a Static Type System

Abadi et al. [1] first formalized adding a `Dynamic` type and a `typecase` construct to a statically-typed language, though programmers had been using such code (often without the benefit of static type-checking guarantees) for some time. This approach is known as incremental, gradual, or hybrid typing.

The key idea is that part of the program is statically type-checked, and part of the program uses a `Dynamic` type. Type errors cannot be the fault of the statically-typed portion, but can be due to the dynamically-typed portion or the boundary between them. The boundary between the parts can be arbitrary—it may lie along module interfaces, or may separate two arguments to an operation. The programmer explicitly decides the boundary, and indicates it by writing type annotations. The key challenge taken up by researchers is defining behavior at the boundary: correctness (no type errors in the static portion), efficiency, and blame control (finding the root cause of an error that arises dynamically in the static portion).

Optional typing (sometimes called pluggable typing) offers no compile-time or runtime guarantees [9, 8], but the types nonetheless offer software engineering and performance benefits.

Gradual typing [21, 36, 38, 37] guarantees that in a fully-annotated program, all type errors are caught at compile time and no dynamic checks are needed. Otherwise, there are checks at the boundary, and objects carry their types with them, at least for higher-order types. The type system replaces type equality with type consistency, which permits coercions that add and remove instances of `?` (their name for `Dynamic`). Quasi-Static Typing [43] is a similar idea, but type errors can occur at runtime, even in a completely annotated program. Another precursor is the BabyJ dialect of JavaScript, which focuses on nominal types.

Thorn's "like types" [7, 47] are intermediate between statically-checked types and `Dynamic`, and their benefits are also intermediate between the two. A like type's uses must be correct according to the static type system, but any value may be assigned to a like type (and must be checked at runtime). Introducing a like type can introduce both compile-time and runtime errors, but does not guarantee the absence of either. A Thorn programmer must manage boundaries between `Dynamic`, like types, and concrete types, rather than just between `Dynamic` and concrete types. Runtime checks at the boundaries guarantee that there are no type errors in the statically-typed portion of the code, but Thorn omits blame assignment, to avoid performance penalties.

Typed Scheme [44, 45] adds procedure declarations and contracts [15, 20] (runtime checks) at the boundary.

Hybrid typing [17, 24] (along with work in a similar spirit by Ou et al. [31]) takes contracts even further. Its refinement types are arbitrary assertions, and the static analysis does both type-checking and theorem-proving to statically discharge as many as possible. A type-checking error arises only from an assertion that can be proved to always fail. Any assertion that cannot be discharged is left in the generated code, to be checked at runtime. There is no guarantee that an assertion failure will not halt the program at runtime.

Blame assignment [16, 44, 45, 18, 46] aims to help programmers understand the root cause of a runtime error. The runtime error may arise in type-checked code, but may be due to an earlier problem in unchecked code, such as putting a value of incorrect type in a data structure. To test higher-order values, blame assignment uses wrappers that test the values at the point of use rather than at the boundary into typed code [15, 2]. Then, a challenge is to improve efficiency (for instance, using fewer wrappers and retaining tail call optimization) [21, 38, 40].

5.3 Comparison to the Ductile Approach

Our work differs from the related work in supporting full, sound static typing throughout a codebase, and permitting the entire codebase to be treated as dynamically typed. Unlike the work of Section 5.1, DuctileJ does no type inference, but requires explicit type annotations. Unlike the work of Section 5.2, DuctileJ makes no guarantees that code in which static types are written will suffer no type error at runtime (if the whole codebase is statically type-correct, the guarantee holds). As a result, DuctileJ can run more code, and the programmer is not forced to explicitly identify and maintain the boundary between the statically and dynamically typed code. We now identify some other differences.

Which types should be dynamic? Most of the related work assumes that a programmer is able and willing to determine which parts of his or her program should be dynamic, or that sound static typing is not a requirement. DuctileJ automatically uses the `Dynamic` type everywhere, even while permitting a programmer to write and check types.

When should dynamic typing be used? Some of the related work assumes that a programmer will want to use dynamic typing indefinitely in some parts of a program; the work seeks interoperability with statically-typed code. Other work aims to evolve dynamically-typed programs into statically-typed programs [5, 9, 39, 44, 45, 41, 7, 47]. But, once the program has been converted, the benefits of dynamic typing are lost during later maintenance stages (unless the programmer explicitly changes types back to `Dynamic`). Our work is unique in its focus on transitioning in both directions of the typing discipline, which supports important software development needs. We also advocate writing (approximations to) static types from the beginning. Even if they are imperfect (e.g., the type used might not yet be defined), they convey intent and cause the programmer to think about static types throughout the interactive design process. They give incremental typing benefits without lock-in, and they ease the final transition to types, making it more palatable and likely.

What is the starting point for the language design? Unlike most related work, we start from the context of a statically-typed language and seek to add the benefits of dynamic typing, rather than vice versa. And, our work has a robust implementation in the context of a real industrial-strength language, giving it the potential to yield practically interesting questions and answers.

What are the scarce resources? Today, computation is plentiful and cheap. Human effort and attention dominate the cost of most software produced today, especially if one includes the costs of bugs and rework that result from inadequate or poorly-focused human attention. Much of the related work is motivated by efficiency, even to the point of removing features like blame assignment [47]. By contrast, our primary focus is on the programmer. We do not force the programmer to write perfect type annotations before running the code. We could utilize optimization approaches proposed by others, but they may be less necessary in our context, since DuctileJ's dynamic execution is used only in the context of in-house testing: dynamism is removed before a release build is produced. We believe it is critical to first determine whether the approaches proposed by us and others are useful. If so, then they can be optimized, rather than the other way around.

5.4 Tolerating Inconsistencies

DuctileJ's key idea is to provide feedback, despite the existence of inconsistencies that will eventually be resolved. This is not a radical idea; a similar approach has been independently proposed[1], but not implemented nor evaluated. The approach is in widespread use informally, as programmers use static checkers (from lint to pluggable type systems) that can be temporarily disabled or simply not run while the programmer focuses on other issues.

Ossher et al. [30] take a similar approach to requirement engineering, providing a single toolset that supports both checked and unchecked modeling. They had found that current tools' consistency checking got in the way and caused users to abandon those tools, losing other features such as change propagation and information migration to downstream tools.

[1]http://faculty.washington.edu/ajko/musings.shtml#typing, http://steve-yegge.blogspot.com/2008/05/dynamic-languages-strike-back.html

DuctileJ relaxes a particular set of compiler requirements, making it possible to run code that violates Java's static type system. Perhaps the sweet spot is to relax the static requirements even more, such as by eliminating checks for uninitialized variables or by executing calls with an incorrect number of arguments. Perhaps the sweet spot relaxes fewer static requirements, such as by prohibiting misspelled identifiers, which were the main annoyance in our case studies. (Smalltalk takes this approach: it is dynamically typed, but when it encounters a symbol that is not in scope, the IDE pops up an offer to declare it.) Or, perhaps the best programming methodology requires stricter type-checking than current languages, or even that code should be forbidden from being tested until after it has been formally proved correct. The particular choices made in our current DuctileJ implementation seem to be effective, based on the case studies. Now that we have proposed the approach, future work should determine whether other choices are even better, and whether particular development styles or problem domains affect the choice.

6. CONCLUSION

We have proposed a novel approach to integrating features normally found in dynamically-typed scripting languages into statically-typed programming languages. The key idea is to permit a programmer to view a program through the lens of completely dynamic typing or completely static typing, and to switch between these views seamlessly, as often as desired. This approach contrasts with other work that attempts to find a middle ground, and that retains neither safety nor flexibility.

Our goal is to enable programmers to work *faster* than they can with statically-typed languages, and to produce more *reliable* code than they can with dynamically-typed languages. Our approach enables the programmer to obtain either static or dynamic feedback whenever the programmer chooses. This overturns current IDE paradigms, putting the programmer in charge of the analysis tools rather than the analysis tools in charge of the programmer.

Our experiments demonstrate that our approach is sufficiently correct and performant, and that the benefits of dynamism aid programmers during both prototyping and evolution. DuctileJ is publicly available, including source code [14].

7. REFERENCES

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

[2] A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *STOP*, July 2009.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, pages 53–64, Oct. 2007.

[4] D. Ancona, G. Lagorio, and E. Zucca. Type inference for polymorphic methods in Java-like languages. In *Theoretical Computer Science: Proceedings of the 10th Italian Conference on ICTCS '07*, 2007.

[5] C. Anderson and S. Drossopoulou. BabyJ: From object based to class based programming via types. In *WOOD*, pages 53–81, Oct. 2003.

[6] J. Aycock. Aggressive type inference. In *Int'l Python Conf.*, 2000.

[7] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, pages 117–136, Oct. 2009.

[8] G. Bracha. Pluggable type systems. In *RDL*, Oct. 2004.

[9] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, Sep. 1993.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

[11] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Boston, MA, USA, 1975.

[12] P. Camphuijsen, J. Hage, and S. Holdermans. Soft typing PHP. Technical Report UU-CS-2009-004, Department of Information and Computing Sciences, Utrecht University, 2009.

[13] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, June 1991.

[14] DuctileJ website. `http://code.google.com/p/ductilej/`.

[15] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, Oct. 2002.

[16] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE*, pages 229–236, Sep. 2001.

[17] C. Flanagan. Hybrid type checking. In *POPL*, Jan. 2006.

[18] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, pages 283–300, Oct. 2009.

[19] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *SAC*, pages 1859–1866, Mar. 2009.

[20] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA*, pages 231–245, Oct. 2005.

[21] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *TFP*, 2007.

[22] A. Holkner and J. Harland. Evaluating the dynamic behavior of Python applications. In *ACSC*, 2009.

[23] JRuby. `http://kenai.com/projects/jruby/pages/JRubyInternalDesign`, July 09, 2010.

[24] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and `Dynamic`. In *SFP*, Sep. 2006.

[25] G. Lagorio and E. Zucca. Just: Safe unknown types in Java-like languages. *J. Object Tech.*, 6(2):71–100, 2007.

[26] P. McGachey, A. L. Hosking, and J. E. B. Moss. Pervasive load-time transformation for transparently distributed Java. In *ByteCode 2009*, Mar. 2009.

[27] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297, May 2009.

[28] D. A. Norman. The "problem" with automation: Inappropriate feedback and interaction, not "over-automation". *Phil. Trans. Royal Soc., B*, 327(1241):585–593, 1990.

[29] S.-O. Nyström. A soft-typing system for Erlang. In *ERLANG '03*, pages 56–71, Aug. 2003.

[30] H. Ossher et al. Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. In *Onward!*, pages 848–864, Oct. 2010.

[31] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *TCS*, pages 437–450, Toulouse, Aug. 2004.

[32] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, June 2010.

[33] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *VMIL*, pages 1–11, 2009.

[34] M. Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, MIT Dept. of EECS, May 2004.

[35] D. Shasha and C. Lazere. *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists*. Copernicus Books, 1998.

[36] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, Aug. 2007.

[37] J. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *DLS*, Pathos, Cyprus, July 2008.

[38] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *ESOP*, Mar. 2009.

[39] J. G. Siek and W. Taha. Gradual typing for functional languages. In *SFP*, pages 81–92, Sep. 2006.

[40] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL*, Jan. 2010.

[41] *STOP*, July 2009.

[42] M. Tatsubori. Living with reflection: Towards coexistence of program transformation by middleware and reflection in Java applications. In *PPL*, Mar. 2004.

[43] S. Thatte. Quasi-static typing. In *POPL*, pages 367–381, Jan. 1990.

[44] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *DLS*, Oct. 2006.

[45] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, Jan. 2008.

[46] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP*, pages 1–16, Mar. 2009.

[47] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL*, Jan. 2010.