ScalaDyno: Making Name Resolution and Type Checking Fault-Tolerant

Cédric Bastin Vlad Ureche Martin Odersky
EPFL, Switzerland
{firstname.lastname}@epfl.ch

Abstract

The ScalaDyno compiler[1] plugin allows fast prototyping with the Scala programming language, in a way that combines the benefits of both statically and dynamically typed languages. Static name resolution and type checking prevent partially-correct code from being compiled and executed. Yet, allowing programmers to test critical paths in a program without worrying about the consistency of the entire code base is crucial to fast prototyping and agile development. This is where ScalaDyno comes in: it allows partially-correct programs to be compiled and executed, while shifting compile-time errors to program runtime.

The key insight in ScalaDyno is that name and type errors affect limited areas of the code, which can be replaced by instructions to output the respective errors at runtime. This allows byte code generation and execution for partially correct programs, thus allowing Python or JavaScript-like fast prototyping in Scala. This is all done without sacrificing name resolution, full type checking and optimizations for the correct parts of the code – they are still performed, but without getting in the way of agile development. Finally, for release code or sensitive refactoring, runtime errors can be disabled, thus allowing full static name resolution and type checking typical of the Scala compiler.

Keywords Scala, dynamic typing, deferred type errors

1. Introduction

In the academic and the professional community, it is agreed that both statically and dynamically typed languages have benefits and drawbacks. In a statically typed programming language the type checker attempts to prove a program is correct up to the constraints encoded by types. This advantageously results in rejecting any program that does not conform, thus ruling out entire classes of runtime errors, such as, for example, calling a method with the wrong number of arguments or with the wrong types. On the contrary, the restrictive type system can get in the way of agile development, since changes need to be reflected in the entire code base to keep its consistency. This forces significant refactoring efforts, with little benefit in terms of prototyping, only to satisfy the type checker.

Dynamically typed languages enable fast prototyping by allowing the programmer to run incomplete prototypes and outputting the errors occurred during execution. However, without a type system and static checks, even the most basic mistakes are discovered only at runtime. This also makes refactoring harder as no tools are available to detect and modify all the related code automatically. The runtime performance of dynamic languages is also generally slower or at least cannot be optimized beyond a specific threshold due to runtime type checks and monkey patching, where any field or method can be added during the execution.

In an ideal programming language, static feedback should be optional, such that the programmer can decide when to use a more dynamic or static approach depending on the development phase (such as bug-fixing, refactoring or preparing for release). Firstly, during bug fixing or new feature development, the programmer might want a more dynamic approach to favor experimentation. However, the correct parts of the code should be compiled as before, without introducing any runtime overhead.

```
1  object Program {
2    // result of running the program: "Hello!"
3    def main(args: Array[String]): Unit = {
4      if (0 == 1)
5        never_called()
6      println("Hello!")
7    }
8
9    // should this method prevent compilation
10   // and execution of the entire program?!?
11   def never_called() = {
12     val x = "Goodbye!"
13     x.noSuchMethod()
14   }
15 }
```

[1] https://github.com/scaladyno/scaladyno-plugin

Scala '14, July 28 – 29 2014, Uppsala, Sweden.

ACM 978-1-4503-2868-5/14/07…$15.00.
http://dx.doi.org/10.1145/2637647.2637649

A typical bug-fixing scenario includes adding an extra argument in several methods. Doing so manually by modifying the signature can lead to an inconsistent code base until all call sites have been manually updated. In this case it would be desirable to be able to execute only the path in the program relevant to the bug, while ignoring overall consistency until the fix is working correctly. In the following example, where the `title` field was added, setting default arguments is not an option, so the programmer needs to proceed with refactoring all method calls:

```
def main(args: Array[String]) {
  val country = "USA"
  if (country == "Germany") {
    greet("Merkel")
    //   ^ expected 2 arguments, received 1
  } else {
    greet("Mr.", "Obama")
  }
}

def greet(title: String, name: String) {
  println(s"Dear $title $name")
}
```

There are two fundamental approaches to address the problem of combining static and dynamic language features. One can either start with a dynamic language and add static checking or start with a static language and make it dynamic. The former is generally impossible due to code patterns such as duck typing and monkey patching which cannot be statically checked. The latter is almost always possible by interpreting the program. Yet, interpretation is slow and cannot accommodate certain features of Scala, such as implicit arguments, which allow filling in the program AST (abstract syntax tree) by reasoning about types and scopes. Therefore simply interpreting Scala code is impossible.

ScalaDyno takes a middle ground approach: it typechecks the program but ignores the parts that are erroneous. This relies on the insight that errors are localized and thus, in many cases, the program can still run correctly despite having erroneous parts. This makes it possible to replace erroneous code by the actual error messages, much like the execution in Python would proceed. The main difficulty here is that later phases of the compiler rely on the code having correctly been type-checked and verified, therefore simply ignoring errors is not enough to compile the program. Instead, ScalaDyno is capable of cleaning up the tree and the symbol table, thus allowing the rest of the compiler phases to proceed:

```
$ ../dy-scalac program.scala
program.scala:13: warning: [scaladyno] value
    noSuchMethod is not a member of String
    x.noSuchMethod()
      ^
one warning found
$ ../dy-scala Program
Hello!
```

However, compiling type correct programs with the ScalaDyno plugin produces the exact same set of instructions as if compiled without. Furthermore executing an incorrect program for a type correct path will yield the same trace as if compiled without the plugin and with all errors fixed.

The contributions of this paper are:

- developing a method to allow fast prototyping in Scala;
- showing how the abstract syntax tree and symbol table consistency can be restored after encountering errors;
- implementing the theory in a compiler plugin of less than 200 lines of code.

2. Approach

Our goal in the ScalaDyno project was to create a modified version of the Scala compiler which allows agile development, especially for prototyping breaking changes to the code base. Such breaking changes include adding or removing fields, methods or classes, changing parameter types or referring to non-existing names. In addition to handling such cases, our solution should not change the semantics of correct code: implicits should still be resolved where possible and the performance of the correct parts of the program should not be impacted. Finally, once the prototyping session is over, all the errors should be available to enable developers to prepare the release.

In our approach we want to collect the errors during compilation, clean the erroneous parts of the tree and replace them by instructions to trigger the error messages at runtime. This makes it possible to compile the erroneous tree down to bytecode and execute it correctly up to the first erroneous instruction.

3. Theory

In order to remove erroneous parts of the AST we have to make some critical assumptions about the types of errors that we can handle. First of all one can note that type errors in Scala, including their side-effects, are localized; they are bound inside a scope defined by a block which might be a method or class body, but the code outside this scope can still be considered correct. However these localized errors can still trigger other (localized) errors in other parts of the code that use the erroneous identifiers, such as instantiations and method calls. In a normal compilation, this should trigger an avalanche of errors in the program.

Yet, in modern compilers, this is not the case: only single, relevant errors are reported to programmers, but the avalanches resulting from these errors are not shown. This has been studied in [10] and is currently being applied in most compilers. In Scala, this is implemented using the `ErrorType`, an alternative bottom type which records the fact that it was produced by an error. Using this marker type combined with specialized typing rules allows the scala compiler to avoid avalanche errors, thus only reporting the relevant error messages.

In the current compiler however, a single compile-time error still halts the entire compilation process. This is be-

cause the later phases in the compiler rely on the assumption that the tree is correct before proceeding. Our approach eliminates the erroneous nodes in the tree and the erroneous symbols created while type-checking the program. This allows later phases to compile the program to bytecode which can be executed.

Yet, the cleaned up code may be missing core parts of its functionality, and thus may produce undesired side effects:

```
1 var path = "/"
2 path = s"/home/$user/.config/myfiles"
3 //             ^ value user undefined
4 sys.run(s"rm -rf $path") // oh noes...
```

The example shows that removing parts of the tree is not enough to guarantee safe execution. What we want is to prevent execution past an erroneous node in the tree. This is why Scaladyno replaces erroneous code by throwing runtime exceptions which prevent further code from being executed. These exceptions contain the exact compiler error message that caused the erroneous tree in the first place. After this cleanup, the rest of the compiler pipeline is able to correctly transform the AST and compile it down to executable bytecode.

Even if some of the type errors have cascading behavior there are still paths through the code that can execute successfully. Allowing these paths to be executed and pass tests is crucial to fast prototyping and refactoring.

4. Implementation

A plugin for the Scala compiler is a separate program that can inject one or more compilation phases and alter the compiler options. In the case of ScalaDyno, we inject a single phase that takes as input the AST from the type checker phase (`typer`), traverses and cleans its erroneous statements by recursively running through all AST nodes. It also cleans up the symbol table by removing any erroneous symbol, namely any symbol whose type is either `ErrorType` or a derivate (e.g. `List[ErrorType]`). The final result is a pruned AST containing only references to correct symbols and a symbol table which only contains correct symbols.

The normal behavior of the name resolution and type checking phases is to issue errors which prevent further compilation of the program. To achieve our goal of allowing partially correct programs to compile, we first need to prevent the compiler built-in `Reporter` from issuing errors which makes further compilation impossible. This can be done by changing the error reporter and transforming errors into warnings. This conversion is however only done for naming and typing errors and not for errors from other phases, e.g. parsing errors as well as overriding and abstract errors which are triggered by the `refchecks` phase and are not currently fixed by ScalaDyno. Since errors are converted to warnings, the programmer already receives some feedback during compilation, in the form of warnings. During reporting, we also record the suppressed errors, which we use to later patch the tree.

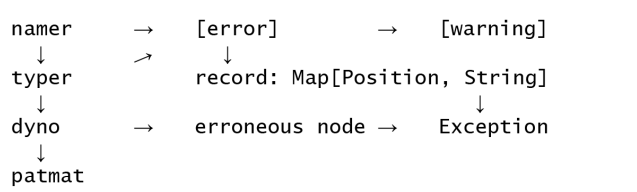

Figure 1. sketch of the comiler plugin and related behaviour

In the type checker, typing errors which happen on some branches in the AST propagate outwards until a stable boundary is reached. Examples of stable boundaries are: the next statement in a block or the next definition in a class, trait, object or package. In order to clean up the tree, we remove the erroneous statements. Yet, as discussed before, we cannot allow the code to execute past an erroneous statement. To implement this, we actually replace erroneous statements by statements which throw exceptions. The message in the exception is the actual error output by the compiler for that particular part of the tree. This is implemented by matching source positions in the tree with source positions of the error messages. Positions are a mechanism by which the compiler records the position of each AST node in the source code. Errors also have positions attached, allowing their messages to point to the exact lines in the source code that triggered them. Therefore, based on the recorded messages and positions and the tree positions we can safely replace the tree nodes by exception-throwing statements.

There are a number of places where simply replacing an erroneous node by a statement doesn't work. Such cases are pattern matches, definitions inside classes, type-defining nodes and annotations. For these cases, we either have special rules which bubble up the statement (in the case of pattern matches) or we issue an error message that we can't properly clean up the tree and abort the compilation. While these errors could be mitigated, the additional complexity significantly burdens the plugin and does not bring significant benefit. Therefore we chose to focus on the most common errors which can easily be cleaned up.

```
1  object Test {
2    def main(args: Array[String]) {
3      val c = Class1(3)
4      val ret =
5        c match {
6          case Class1(1) => "one"
7          case NoSuchClass(2) => "two"
8          case Class1(3) => "three"
9        }
10     println(ret)
11   }
12 }
```

The above code will result in a cleaned-up AST, after the work of the compiler plugin, with the node:

```
val ret: String = sys.error("
examples/Test3compilesMatch.scala:10: not
    found: value NoSuchClass
                    case NoSuchClass(2) => "two"
                         ^


");
```

This translation enables fast prototyping by allowing partially-incorrect code to still compile, but does not allow it to run by throwing an exception that prevents further execution past the erroneous statement. The next section will present the related work.

5. Related work

Several approaches to enabling faster prototyping are currently in use: (1) dynamic languages with checking, (2) reflection, (3) proxies and (4) moving type computations to runtime.

5.1 Dynamic Languages with Checking

A dynamic language can be augmented it with type annotations which can then be used to give static feedback at compile time. These annotations would be optional, and the checks would only trigger if both the actual and the expected type are annotated, as the Dart programming language does [1]. Yet, such approaches are still fundamentally dynamic, as checking all the code would require adding annotations everywhere. Typed Racket [11] as well as Strongtalk (a Smalltalk dialect with optional static typing support) [5], are other examples of brining static typing to a dynamic language.

A pitfall in dynamic languages is allowing patterns such as monkey patching and duck typing, such as, for example, in JavaScript. These patterns, once used, make the code base impossible to statically typecheck, since proving their correctness can, in adversarial cases, require solving the termination problem, which is undecidable. Still there are solutions, such as `like types` [13], which split the work between compile time and run time.

Combining dynamic and static typing is possible with `like types` [13]. With this method one can use either static typing with a nominal type system or dynamic typing using the `dyn` type. However there is a possibility to use static checks for dynamically typed objects as well. To do this an intermediate type structure is introduced where each nominal type gets a corresponding `like` version: `A <: like A`. This means that you can convert an instance of type `dyn` to any `like type`, which, at runtime, triggers an interface conformance test which can abort the execution in case of failure. After the conversion, explicit or implicitly, when used as method parameters, static checks can be done on the new object. However due to the possibly different field and method layout only runtime checks but not resolution can be done on those objects.

5.2 Using Reflection

A second approach is completely switching to the use of reflection, practically turning a statically typed language such as Java or Scala into a dynamic language. This has been implemented in DuctileJ and DuctileScala [4, 8]. Yet this approach makes heavy use of reflection and is unable to resolve implicits. This makes it unsuitable for our use case, as it introduces significant overheads for correct programs and it potentially prevents correct programs that use implicit arguments from running at all.

To add dynamic behavior to the Java programming language, DuctileJ [4] does a detyping transformation before the real typing phase. This detyping consists of converting the types of all the variables and fields, as well as all the method parameters, to `Object`, which is a the Java super class of any other types (the top type). In addition to these transformations, a runtime library `RT` is needed to support late binding:

```
RT.newInstance("ClassName")
RT.select(instance, "fieldName")
RT.invoke("methodName", instanceName, args)
RT.assign(instance, "fieldName", value)
RT.cast("ClassName", instance)
```

Such a transformation however requires the duplication of the typing phase in the compiler, the first to detype the code and collect possible error messages the second to do the standard typing on the modified tree. Also due to method overloading, method signatures needs to be mangled which means that each original parameter needs to be duplicated, one is needed to carry the type, the other to carry the value. Of course this creates some additional problems when working with pre-compiled libraries. However this transformation allows duck-typing which is also considered an important feature of dynamically typed languages.

Very similar to DuctileJ, DuctileScala [8] also introduces a set of new compiler phases (`signature`, `earlynamer`, `earlypackageobjects`, `earlytyper`, `detyper`) which perform detyping as well as other necessary transformation (e.g. signature mangling). Some transformations however are even more complicated due to implicit conversions (views) as well as pattern-matching which are unique features of Scala compared to Java.

5.3 Proxies

A third approach is using an proxy technique, such as the `Dynamic` trait in Scala [2] which acts as a proxy that allows rewriting unresolved methods to more general proxies that can later use reflection to call the correct methods. Such approaches have been used to allow interoperability between Scala and JavaScript [6], but they are not able to handle all cases necessary for prototyping, notably they require correct name and type resolution to work properly. Unfortunately, in many practical scenarios, once an error has occurred, type inference doesn't kick in anymore and is not able to infer the `Dynamic` marker in a value's type, such that methods may be called on it.

Instead of making the entire Scala language more dynamic, the scala-js approach [6] focuses on integrating combined development with Scala an Javascript. A specific set of Scala classes is created which are only facade types for a corresponding Javascript object, those constructs can then either be compiled to dynamic Javascript or to Scala code. Some complications encountered were due to implicit conversions and the connections with existing Javascript libraries.

A somewhat similar approach is taken by scala-virtualized [9]: using the analogy to hardware virtualization for programming languages, one can customize the build-in language constructs by transforming them into method calls:

```
def __ifThenElse[T](cond: Rep[Boolean], thene:
    => Rep[T], elsee: => Rep[T])
```

Scala-Virtualized enables advanced multi-stage programming in Scala, using the Lightweight Modular Staging (LMS) framework. The LMS framework has been very successful at optimizing embedded domain-specific languages (DSLs). One such DSL is JavaScript, which can be embedded in Scala [7]. Gradual typing allows granularity such that external JavaScript libraries do not need to be typed as they are only needed in later phases. In addition the DSL code can either be compiled to JavaScript or be used as Scala which means that computation can either be done on the server side or the client side.

5.4 Deferred Type Errors

The GHC [12] compiler allows the addition of equality proofs to the system FC intermediate language. Equality proofs can be completely erased so that they induce no runtime overhead. They are also first class citizens such that proof-as-values allow to defer type errors to runtime so partially type incorrect programs can compile and execute.

The techniques presented are summarized in Table 1.

	DuctileScala	Haskell	Dynamic / Scala-JS	Scala-Virtualized	Dart	dynamic languages	ScalaDyno
def. name res. errors	✗	✗	✗	✗	✗	✓	✓
def. type errors	✓	✓	✓	✓	✓	✓	✓
fast runtime	✗	✓	✗	✓	✓	✓	✓
compile-time res.	✗	✓	✓	✓	✗	✗	✓

Table 1. Summary of different approaches to deferring type errors. Abbreviations: def. = deferred, res. = resolution

6. Evaluation

To evaluate the plugin we compiled scalaz [3], a well known type class library in Scala. To test the functionality, we commented out class `CobindOps`. We received 4 warnings but the compilation succeeded and we were able to execute some of the scalaz-tests. This transformation does however not work for all the cases due to an incomplete tree cleanup and the fact that parsing and overriding errors can still prevent the code from being compiled all the way.

7. Conclusion

Deferring type errors are a very important feature which enables quick prototyping and development. Most implementations that allow fast prototyping fall short of the goals of optional type checking, maximum performance for correct code and applicability for the Scala language. The prototype we have presented shows promising results with a small development investment of only 200 lines of code.

8. Acknowledgments

The authors would like to thank Damien Engels for his help with setting up the infrastructure, the reviewers who read the initial draft of the paper and gave us useful feedback and the members of the Programming Languages Laboratory in EPFL with whom we had very frutiful discussions on the fundamentals of avoiding type checking errors (and especially to Hubert Plociniczak, Eugene Burmako and Sandro Stucki).

References

[1] Dart Programming Language. URL `https://www.dartlang.org/`.

[2] SIP-17: Type Dynamic. URL `http://docs.scala-lang.org/sips/completed/type-dynamic.html`.

[3] The Scalaz library. URL `https://github.com/scalaz/scalaz`.

[4] M. Bayne, R. Cook, and M. D. Ernst. Always-available static and dynamic feedback. In *ICSE*. ACM, 2011.

[5] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *ACM SIGPLAN Notices*. ACM, 1993.

[6] S. Doeraene. Scala. js: Type-Directed Interoperability with Dynamically Typed Languages. Technical report, 2013.

[7] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript As an Embedded DSL. In *ECOOP*. Springer-Verlag, 2012.

[8] R. Martin, D. Perelman, J. Lei, and B. Burg. Ductilescala: Combined static and dynamic feedback for scala.

[9] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *PEPM*. ACM, 2012.

[10] N. Ramsey. Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions. *The Computer Journal*, 42(5):360–372, 1999.

[11] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL '08*. ACM, 2008.

[12] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. *SIGPLAN Notices*, 2012.

[13] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10*. ACM, 2010.