A Scalable Infrastructure for Teaching Concepts of Programming Languages in Scala with WebLab

An Experience Report

Tim van der Lippe Thomas Smith Daniël Pelsmaeker Eelco Visser

Delft University of Technology

T.J.vanderLippe@student.tudelft.nl, T.N.Smith@student.tudelft.nl, D.A.A.Pelsmaeker@student.tudelft.nl, E.Visser@tudelft.nl

Abstract

In this paper, we report on our experience in teaching a course on concepts of programming languages at TU Delft based on Krishnamurthi's PAPL book with the definitional interpreter approach using Scala as meta-language and using the WebLab learning management system. In particular, we discuss our experience with encoding of definitional interpreters in Scala using case classes, pattern matching, and recursive functions; offering this material in the web-based learning management system WebLab; automated grading and feedback of interpreter submissions using unit tests; testing tests to force students to formulate tests, instead of just implementing interpreters; generation of tests based on a reference implementation to reduce the effort of producing unit tests; and the construction of a product line of interpreters in order to maximize reuse and consistency between reference implementations.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer science education; D.3.4 [*Programming Languages*]: Processors—Interpreters; D.2.5 [*Software Engineering*]: Testing and Debugging—Test generation

Keywords Teaching, Concepts of Programming Languages, Definitional Interpreters, Testing, Scala, WebLab

1. Introduction

There are essentially three approaches to the study of concepts of programming languages. A popular approach is to study several real world programming languages as representative examples of (collections of) concepts [13, 17]. The study activity consists of reading typical example programs and perhaps writing some programs in each language to conduct comparative experiments. This approach requires no other tools than existing programming languages and has the useful side effect of teaching these languages. However, the approach lacks depth and precision; students only get an understanding of concepts through their observable behavior in experiments.

An approach that addresses this concern is to study the formal semantics of programming languages. For example, Nielson and Nielson [7] present the formal specification of languages using denotational, operational, and axiomatic semantics. This provides a precise description of the meaning of a language and supports formal reasoning about applications such as program analysis. The recent work of Pierce et al. [10] transforms this approach from paper formalization to mechanized formalization with proof assistants. The downside of this approach is that it requires the introduction of a fair amount of heavyweight theoretical machinery.

The third approach provides a middle ground between these approaches by using *definitional interpreters* as its main vehicle. A definitional interpreter, as introduced by Reynolds [11, 12], is a program that interprets an abstract syntax tree representation of a program and computes its value. This makes explicit the mechanisms behind language constructs, or at least abstractions of such mechanisms. It allows to directly study the operational behavior of language constructs and to study the effect of alternative semantics by means of experiments. This approach combines the experimental style of the first approach with the precision of the second without introducing more machinery than that of a basic functional language. The approach was introduced as a tool for teaching programming languages by Kamin in his book "Programming languages — an interpreter-based approach" [2]. Krishnamurthi has adopted the approach, first using Scheme as meta-language and object language [4], more recently using the Pyret teaching language as meta-language and a Scheme-like language as object language [5].

In this paper, we report on our experience applying the definitional interpreter approach in a course on concepts of programming languages at TU Delft using Scala as meta-

SCALA'16, October 30–31, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4648-1/16/10...$15.00
http://dx.doi.org/10.1145/2998392.2998402

language and using the WebLab learning management system [3, 15]. The course is based on Krishnamurthi's PAPL book [5]. In our course we chose to adopt Scala instead of Pyret as the implementation language. Scala provides similar features for algebraic data type definition, pattern matching, and immutable data types that simplify programming language parsing and interpretation. Using Scala introduces our students to functional programming on a platform they are familiar with. And it also helps that there are more Scala resources for the students to consult. (The existing support for Scala in WebLab also contributed to the choice of language.)

The main study activity in the course is writing interpreters. However, just submitting the code of interpreters makes grading a significant effort (especially considering the increasing enrolment into the curriculum) and does not encourage students to put additional effort in developing examples and test cases for the languages they implement. To address these concerns we have developed infrastructure for automatically assessing student submissions and providing feedback at scale *and* such that the effort of developing assignments and their automated grading is manageable. In particular, we discuss our experience with

- encoding of definitional interpreters in Scala using case classes, pattern matching, and recursive functions (Section 2);
- offering this material in the web-based learning management system WebLab (Section 3);
- automated grading and feedback of interpreter submissions using unit tests (Section 4);
- generation of tests based on a reference implementation to reduce the effort of producing unit tests (Section 5);
- testing tests to encourage students to formulate tests, instead of just implementing interpreters (Section 6); and
- the construction of a product line of interpreters in order to maximize reuse and consistency between reference implementations (Section 7).

We expect that this report will be helpful for instructors considering to apply the approach in their courses.

2. Definitional Interpreters in Scala

The course revolves around developing interpreters for a range of small languages with representative language constructs. In this section we illustrate the approach by means of an interpreter for a small language with arithmetic and functions as first-class citizens, which we will use as the running example in the rest of the paper.

Abstract Syntax Rather than putting much emphasis on syntax and parsing, PAPL[1] makes the abstract syntax the central representation in terms of which semantics is studied.

[1] We will attribute the approach to the book [5] using its acronym PAPL rather than to its author(s) and omit further citation.

```
sealed abstract class Ext
// Numbers
case class NumExt(num: Int) extends Ext
// Binary and unary operators. E.g.: (+ 1 3)
case class BinOpExt(s: String, l: Ext, r: Ext)
  extends Ext
case class UnOpExt(s: String, e: Ext) extends Ext
// Application of a function with one argument
case class AppExt(f: Ext, a: Ext) extends Ext
// Definition of a function with one argument
case class FdExt(arg: String, body: Ext) extends Ext
// Variables
case class IdExt(c: String) extends Ext
```

Figure 1. Abstract syntax of extended (sugared) language.

```
sealed abstract class Core
case class NumC(num: Int) extends Core
// Numerical addition and multiplication
case class PlusC(l: Core, r: Core) extends Core
case class MultC(l: Core, r: Core) extends Core
// Function application, definition and variables
case class AppC(f: Core, a: Core) extends Core
case class FdC(arg: String, body: Core) extends Core
case class IdC(c: String) extends Core
```

Figure 2. Abstract syntax of core language.

```
sealed abstract class Value
case class NumV(n: Int) extends Value
// Closures to support first-class functions
case class ClosV(f: FdC, e: List[Bind]) extends Value

// For binding a value to a variable
case class Bind(name: String, value: Value)
```

Figure 3. Representation of values and bindings.

Abstract syntax trees are represented using algebraic data types, which are conveniently defined using case classes in Scala.

PAPL explicitly introduces the distinction between a feature rich (sugared) source language and a minimal core language that includes the essential constructs to achieve a certain level of expressiveness. The extended version of our example language is defined in Figure 1. It features number literals, generic binary and unary operators, unary function literals (lambdas), function application, and variables.

Rather than taking the core language to be a subset of the extended source language, PAPL defines it as a separate data type. The core for our example language is defined in Figure 2. It is mostly the same as the extended language, but instead of a generic representation for operators, it has explicit representations for an addition and multiplication operator only.

Parsing Since writing object programs as abstract syntax trees is tedious, a little concrete syntax is useful. To avoid the overhead of implementing full blown parsers, PAPL uses S-expressions as a concrete syntax substrate. For example, here is an expression in our example language:

```
object Parser {
  def parse(sexpr: SExpr): Ext = sexpr match {
    case SNum(n) => NumExt(n)
    case SList(List(SSym("lambda"),
            SList(List(SSym(arg))), body)) =>
      FdExt(arg, parse(body))
    case SList(List(SSym(sym), l, r))
      if Set("+", "-", "*").contains(sym) =>
      BinOpExt(sym, parse(l), parse(r))
    case SList(List(SSym(sym), e))
      if Set("-").contains(sym) =>
      UnOpExt(sym, parse(e))
    case SList(head :: arg :: Nil) =>
      AppExt(parse(head), parse(arg))
    case SSym(s) => IdExt(s)
  }
  def parse(str: String): Ext = parse(Reader.read(str))
}
```

Figure 4. Parser

```
object Desugar {
  def desugar(e: Ext): Core = e match {
    case NumExt(n)   => NumC(n)
    case BinOpExt(op, l, r) => op match {
        case "+" => PlusC(desugar(l), desugar(r))
        case "*" => MultC(desugar(l), desugar(r))
        case "-" => PlusC(desugar(l), MultC(NumC(-1),
                              desugar(r)))
      }
    case UnOpExt(op, e) => op match {
        case "-" => MultC(NumC(-1), desugar(e))
      }
    case AppExt(f, a) => AppC(desugar(f), desugar(a))
    case IdExt(c) => IdC(c)
    case FdExt(arg, body) => FdC(arg, desugar(body))
  }
}
```

Figure 5. Desugarer

```
  ((lambda (x) (* 2 (+ x (- 2 1)))) 8)
```

This divides the task of parsing into application of a generic `read` function that parses a string representation of a concrete syntax S-expressions into an object of the `SExpr` data type, and a language-specific `parse` function that translates an `SExpr` object into an AST object. We provide the `read` function and `SExpr` data type to our students who only have to write the `parse` function. (And in later assignments we even give them the `parse` function.)

With this approach parsing is reduced to matching S-expression patterns corresponding to constructs of the language as illustrated in Figure 4. For example, the pattern

```
  SList(List(SSym("lambda"),
             SList(List(SSym(arg))),
             body))
```

matches the S-expression

```
  (lambda (<arg>) <body>)
```

where `<arg>` is the name of the function argument and `<body>` should be an expression.

```
object Interp {
  def interp(e: Core, env: List[Bind]): Value =
    e match {
      case NumC(n) => NumV(n)
      case PlusC(l, r) =>
        val (NumV(vL), NumV(vR)) =
          (interp(l, env), interp(r, env))
        NumV(vL + vR)
      case MultC(l, r) =>
        val (NumV(vL), NumV(vR)) =
          (interp(l, env), interp(r, env))
        NumV(vL * vR)
      case AppC(f, a) =>
        val ClosV(FdC(arg, body), env_clos) =
          interp(f, env)
        val argVal = interp(a, env)
        interp(body, Bind(arg, argVal) :: env_clos)
      case IdC(s) => lookup(s, env)
      case fdc@FdC(arg, body) => ClosV(fdc, env)
    }
  def lookup(x: String, nv: Environment): Value =
    nv match {
      case List() => throw InterpException()
      case Bind(y, v) :: nv2 =>
          if (x == y) v else lookup(x, nv2)
    }
}
```

Figure 6. Interpreter

Desugaring A core language expresses the key computational ideas. However, programming languages often provide constructs that make programming more convenient, but can be expressed in terms of the core language. Rather than just ignoring such syntactic sugar, PAPL introduces an explicit desugaring step in the semantic pipeline.

The `desugar` function mostly copies constructs in the extended language to their counterparts in the core language, and in the process translates sugar patterns to combinations of the constructs in the core language. Figure 5 illustrates this for our example language, transforming `Ext` objects into `Core` objects. The function translates generic binary operators to the specific operators of the core language. The addition and multiplication operators are translated directly, but the binary and unary minus operators are desugared in terms of the former.

Interpreter The key component of the semantic pipeline is the interpreter. The `interp` function computes the value of an expression in the core language, i.e. it is a function from objects of type `Core` to objects of type `Value`. Just like abstract syntax trees, we use case classes for the representation of value objects. Figure 3 defines as values for our example language numbers (`NumV`), the values of arithmetic operations, and closures (`ClosV`), the values of function expressions. To interpret our example language, the interpreter also requires an environment to keep track of the binding of values to variables. The `Bind` class in Figure 3 represents

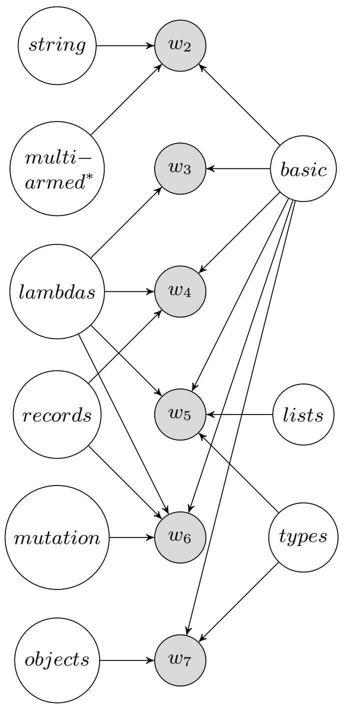
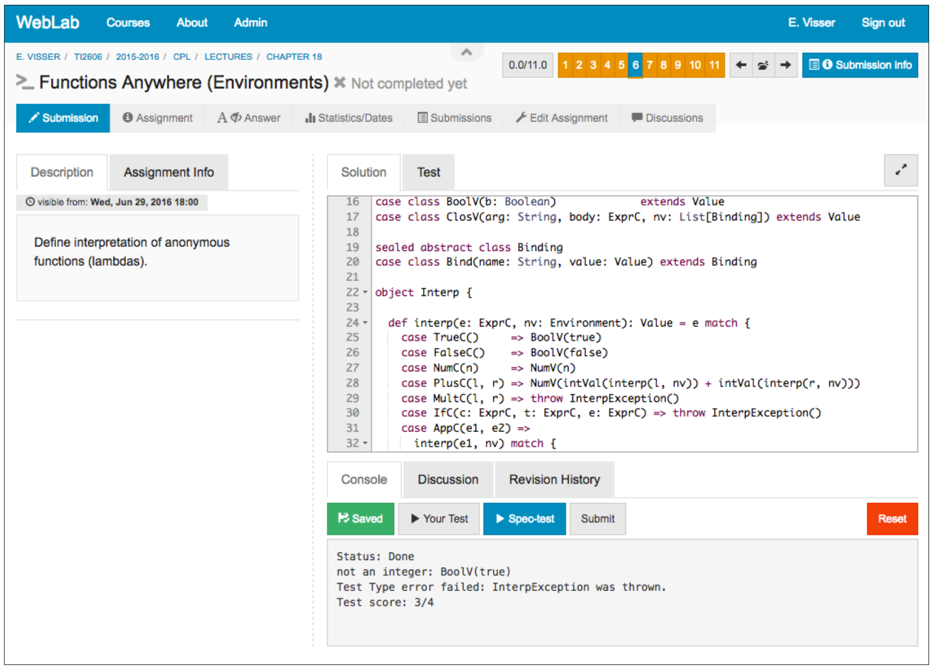


Figure 7. Overview of languages.



Functions Anywhere (Environments) ✖ Not completed yet

visible from: **Wed, Jun 29, 2016 18:00**

Define interpretation of anonymous functions (lambdas).

```
case class BoolV(b: Boolean)                  extends Value
case class ClosV(arg: String, body: ExprC, nv: List[Binding]) extends Value

sealed abstract class Binding
case class Bind(name: String, value: Value) extends Binding

object Interp {

  def interp(e: ExprC, nv: Environment): Value = e match {
    case TrueC()      => BoolV(true)
    case FalseC()     => BoolV(false)
    case NumC(n)      => NumV(n)
    case PlusC(l, r) => NumV(intVal(interp(l, nv)) + intVal(interp(r, nv)))
    case MultC(l, r) => throw InterpException()
    case IfC(c: ExprC, t: ExprC, e: ExprC) => throw InterpException()
    case AppC(e1, e2) =>
      interp(e1, nv) match {
```

```
Status: Done
not an integer: BoolV(true)
Test Type error failed: InterpException was thrown.
Test score: 3/4
```

Figure 8. WebLab user interface.

such bindings, and an environment consists of a list of such bindings. A function value should keep track of the bindings at the point of its definition. Therefore, a closure has an environment as argument in addition to the abstract syntax tree of the function.

A syntax-directed interpreter is defined by induction on the structure of the abstract syntax tree, defining a match case for each language constructor, recursively invoking the interpreter on sub-trees. Figure 6 defines the interpreter for our example language. The interesting case for this language is the evaluation of function expressions, function application, and variables. A variable is evaluated by looking up its binding in the current environment. A function expression returns a closure with the definition-time environment. A function application evaluates the body of a function value in an environment, extended with a binding of the value of the actual parameter to the formal parameter. The crucial point of this definition is the treatment of environments in function application. In order to realize static scoping semantics, the actual parameter should be evaluated in the call-time environment, but the function body should be evaluated in the definition-time environment from the closure. Having an executable definition of this semantics allows direct experimentation with alternatives. Typical mistakes made by students are evaluating the body in the call-time environment, giving dynamic scoping, or evaluating the actual parameter in the closure environment. More intricate errors emerge when extending the language to functions with multiple parameters.

2.1 Course Organization

In the course we extend and modify this language to study more concepts of programming languages. Over the span of the course, students write a parser, desugarer, and interpreter for a new language each week. The lab covers the following weekly topics:

1. Scala introduction: basic functional programming and test driven development in Scala
2. Arithmetic and Booleans: architecture of the parse-desugar-interpret approach
3. First-class functions: names, environments, function values, closures, function application
4. Records: extensible, immutable records
5. Type checking: type checking for a language with lists, type soundness
6. Mutation: boxes, mutable variables, stores
7. Mini Java: small object-oriented programming language
8. Type inference: type expressions, unification

Figure 7 shows the dependency graph that illustrates which interpreters were used in which week of the 2015–2016 edition of the course. For example, in week 5 (w_5), students extend the basic language (created as assignment in week 3 (w_3)) with type-checking and lists.

3. WebLab

The enrollment into the curriculum has been steadily increasing. In the 2015–2016 edition, 180 students enrolled into the second year course, and this number is expected to further increase in the coming years. While not near MOOC scale, these numbers already create a significant grading and administration effort. Furthermore, the 10 week quarter in Delft requires weekly deadlines for assignments and a very short turnaround time for grading and feedback.

To scale our education without involving large numbers of teaching assistants we have been developing WebLab[2], a web-based learning management system especially geared to programming education [3, 15], including support for Scala. The system integrates the development of assignments by instructors, writing submissions by students, and the administration of results for an entire course.

Interface Figure 8 shows the user interface for developing a submission to a programming assignment, with the following components: An editor (under the 'Solution' tab) for developing the solution to the assignment. An additional editor (under the 'Test' tab) for developing unit tests to test the solution. The 'Your Test' button for invoking the tests written in the 'Test' tab. The 'Spec Test' button for invoking the secret specification tests. A console for displaying feedback from test invocation. A revision history for keeping track of all edits to the program and tests. A discussion tab for asking questions to the teaching assistants.

The execution of a solution against a test set is done on the server and may include compilation for compiled languages. We strive to give immediate feedback on execution, which requires execution within seconds of invocation. To that purpose, the LabBack back-end provides a pool of running JVMs to execute test jobs [15]. LabBack currently supports Scala, Java, Python (via JPython), JavaScript, and C (via clang produced JavaScript). A benefit of this set-up is that we do not need to worry about deployment of programming environments on student machines.

Automated Grading The key benefit of WebLab is that it integrates course administration with submission, persistent storage, and grading of student solutions. WebLab organizes all assignments of a course in a tree structure with configurable grading schemes at each node of the tree. Grades for individual assignments are aggregated into grades for composite assignments according to a configurable weighted grading scheme.

Programming assignment submissions can be graded using two mechanisms. First, a submission is tested against a set of secret *specification tests*. Students do not see these tests, nor their output on failure. However, students do see the ratio of successful versus all scores of each specification test run (e.g. 22/30). Automated grading is useful for instructors as it considerably lowers to effort of grading programming submissions. It is an advantage to students as well, since the testing ratio feedback, even if minimal, provides very early feedback on progress. Second, a submission can be scored against a rubric that states Boolean criteria that it should satisfy. Where needed, rubric grading can address grading issues that cannot be covered by unit testing. While this requires teaching assistant intervention, the WebLab workflow reduces the grading effort since basic testing (does it compile? is it functionally correct?) is taken care of by the specification test. The weight of each grading mechanisms can be configured for each assignment.

Related Work With the rise of MOOCs the issue of scalable courses has been addressed by others as well. For example, Miller et al. [6] describe the set-up for a course on functional programming principles in Scala, using cloud-based computing to grade the style and implementation of student submissions. However, they require that students install Scala, the Scala build tool *sbt*, and an IDE to get started. WebLab instead provides an online code editor, which not only eases the requirements on the students and their systems, but also ensures that all students work in the same environment with the same software.

Course Development Effort WebLab reduces grading effort and speeds up feedback to students. This frees up teaching assistants to help students during assisted labs, rather than spending time on grading. However, this requires a careful upfront design and development of the assignments, including high quality test sets. In the remainder of this paper we elaborate on the Scala infrastructure we developed for also reducing this upfront design and development cost.

4. Testing Interpreters

The default interface of WebLab for automated grading is based on unit testing, taking the ratio of successful tests to the size of the test suite as measure for a grade. For Scala, WebLab provides a binding to the ScalaTest unit testing framework [14]. Thus, to check correctness of student submissions of parsers, desugarers, and interpreters we developed specification test suites using ScalaTest. Figure 9 shows an example test suite with integration tests that exercise the whole chain of parsing, desugaring, and interpretation. The result is either a value in the case of a positive test or an exception in the case of a negative test. However, this basic approach is too simplistic, and needs to be refined, in order to test that all components of the chain are implemented correctly. This also provides us with better diagnostics to determine the errors made by a student. Thus, for a single input program, we write separate assertions for each component.

Figure 10 shows how the first, positive, test of Figure 9 is rendered. The first assertion tests that the parser produces the expected AST in the extended language. The third assertion tests the interpretation of the corresponding core language AST. The second assertion tests desugaring. Since there are many possible equivalent desugarings for each sugared expression, testing the AST produced by desugaring requires

[2] `https://weblab.tudelft.nl`

students to guess the desugaring our tests checks for. Therefore, we check the result of interpreting the result of desugaring. Figure 11 shows the rendering of the second, negative, test of Figure 9. In order to check that an exception is thrown at the correct stage, we verify that the preceding stages execute successfully.

Clustering Tests The specification tests are used to provide feedback to students about the quality of their solution and is used to calculate a grade. A student can continuously run the specification tests on their solution to get the number of successful tests versus the total number of tests. This serves as a useful indicator of the completeness of their solution. WebLab uses the ratio of success versus failed specification tests to determine the student's grade. A disadvantage of this approach is that all tests contribute equally to the grade, while not all tests are equally important. For example, we include tests that check that the solution for aspects corresponding to the previous assignment still works, but we do not want these tests to count very much towards the final grade. A related issue is that tests may be dependent. For example, negative tests for the corner cases of the addition operator should not succeed when the positive test for the addition operator does not succeed.

To have more control over the test dependencies and the contribution of tests to the grade, we group tests into *clusters*. A cluster of tests is counted as a single test in Scala, and all tests in a cluster must succeed for the cluster to succeed. By varying the number of clusters, we can influence the test success ratio, and therefore the grade.

A cluster of only negative tests will succeed even when the feature in question has not been implemented. To prevent this, each cluster must have at least one positive test for that feature.

To get even better control of the grade, clusters have percentual weights. The example in Figure 12 shows two clusters with tests for the Week 2 assignment as used in the specification tests of Week 3. The total weight the previous week is 30%. This week has 2 clusters, one has weight 1, the other weight 2. As a result, the first cluster increases the test score by $1 \cdot 30/3 = 10$, the second by $2 \cdot 30/3 = 20$. At the start of a test, the `totalScore` is increased with the corresponding total attribution value. The `achievedScore` is increased with the same attribution value if and only if all asserts in the tests have succeeded. The student does not receive any credit if only a portion of the tests succeed.

5. Test Generation

Writing test suites following the approach of Section 4 is extremely tedious. To avoid this tedium we have developed an internal test definition DSL. The DSL reduces the specification of a test case to the input program to be evaluated. For example, Figure 13 defines a small test suite consisting of two groups, each consisting of clusters of positive and negative tests. A test is represented as an instance of a case class

```
class TestSpec extends FunSuite {
  test("POS: (+ 4 5)") {
    assertResult(NumV(9)){
      Interp.interp(
        Desugar.desugar(
          Parser.parse("""(+ 4 5)""")
        )
      )
    }
  }
  test("NEG: ((lambda () 13) (+ 4 5))") {
    intercept[InterpException] {
      Interp.interp(
        Desugar.desugar(
          Parser.parse("""((lambda () 13) (+ 4 5))""")
        )
      )
    }
  }
}
```

Figure 9. Example test suite.

```
test("POS: (+ 4 5)") {
  assertResult(BinOpExt("+", NumExt(4), NumExt(5))){
    Parser.parse("""(+ 4 5)""")
  }
  assertResult(NumV(9)){
    Interp.interp(
      Desugar.desugar(
        BinOpExt("+", NumExt(4), NumExt(5))
      )
    )
  }
  assertResult(NumV(9)){
    Interp.interp(PlusC(NumC(4), NumC(5)))
  }
}
```

Figure 10. An expanded positive test.

```
test("NEG: ((lambda () 13) (+ 4 5))") {
  assertResult(AppExt(FdExt(List(), NumExt(13)),
    List(BinOpExt("+", NumExt(4), NumExt(5))))){
      Parser.parse("""((lambda () 13) (+ 4 5))""")
  }
  Desugar.desugar(AppExt(FdExt(List(), NumExt(13)),
    List(BinOpExt("+", NumExt(4), NumExt(5)))))
  // Verify an exception is thrown in the interpreter
  intercept[InterpException] {
    Interp.interp(AppC(FdC(List(), NumC(13)),
      List(PlusC(NumC(4), NumC(5)))))
  }
}
```

Figure 11. An expanded negative test.

that records the test program's string representation, whether it is a positive or negative test, and optionally other relevant data such as the binding environment. The test generator, defined in Figure 14, uses this representation in a two-stage process to generate the ScalaTest code that can be used in WebLab. The first stage calls the reference implementation

```
var totalScore : Double = 0
var achievedScore : Double = 0

// Normalize to actual score, printed out to student
def getWeightedScore: Double = achievedScore/totalScore

/********************
 * Week 2           *
 * Total Score: 30.0 *
 ********************/
test("Week 2: Binary operators argument arity") {
  // Cluster score weight: 1.0
  totalScore += 10
  // NEG: (+ 4 5 6)
  intercept[ParseException] {
    Parser.parse("""(+ 4 5 6)""")
  }
  // If the cluster passes all assertions,
  // this last line will be reached
  achievedScore += 10
}
test("Week 2: arithmetic good") {
  // Cluster score weight: 2.0
  totalScore += 20
  // POS: 2445
  assertResult(NumExt(2445)){
    Parser.parse("""2445""")
  }
  assertResult(NumV(2445)){
    Interp.interp(Desugar.desugar(NumExt(2445)))
  }
  assertResult(NumV(2445)){
    Interp.interp(NumC(2445))
  }
  // If the cluster passes all assertions,
  // this last line will be reached
  achievedScore += 20
}
```

Figure 12. Example clusters with weighted scores.

to produce the expected parsed AST, desugared AST, and value produced by interpretation. The second stage generates a sequence of statements that will perform this same process on student implementations and asserts that it gives the same result as our reference implementation.

The core class for test generation is the `TestSpecBase` class, which is an abstract class that has all the boilerplate code for generating ScalaTest tests (Figure 14). Derived from this is the abstract `TestSpec` class, which defines sets of tests that are shared between test specifications for each chapter. Finally, for each assignment a class is derived from the `TestSpec` with only those tests that apply to the solution of that assignment. This keeps the tests clearly separated. Thanks to the test generator we can easily write even more complex tests. For example, the test in Figure 15 checks that application arguments are *not* evaluated in the environment of the closure being applied. Figure 16 shows the corresponding generated code.

The actual code generation is done through the virtual `TestSpecBase.generateTestCase()` method. It produces a Scala source file with the generated test case code substituted into it. The positive and negative test cases for the generator are represented as case classes. By default the `TestSpec` class defines and handles two case classes `Pos` and `Neg` for positive and negative respectively that test the parser, desugarer and interpreter. If a particular chapter requires more case classes than the default, we can add them in the subclass and override the `generateTestCase()` method to handle those case classes too. The test case classes usually specify only the syntax and any environment in which the test must be run. The test case's syntax is then fed through our own parser, desugarer and interpreter to find what kind of AST or result it produces. Then Scala tests are generated that check that the student solution produces the same results. For a negative test it is asserted that the test fails with a particular exception.

```
object Week3 extends TestSpecBase {
  def getTests = List(
    Group("Week 2", scoreTotal = 30,
      CoreLanguageTests.parserTests
       ++ CoreLanguageTests.clusters),
    Group("Arithmetic", scoreTotal = 70, List(
      Cluster("Binary operators arity", List(
        Neg("(+ 4 5 6)"),
        Neg("(+ 4)"),
        Pos("(+ 4 5)")
      )),
      Cluster("Arithmetic tests", List(
        Pos("(+ 248 80)"),
        Pos("(+ (+ 12 55) 89)"),
      ))
    ))
  )
}
```

Figure 13. Example test suite in our internal test DSL.

6. Testing Tests

Writing a correct interpreter for a language is only half of the job of studying semantics. It is important to understand how an interpreter can go wrong. That is, to understand what is *not* correct behavior. This requires developing test cases for corner cases by thinking through feature interactions. For example, when all test cases in a test suite use distinct variables, it will not discover interpreters with variable capture (dynamic scoping) errors.

In a previous edition of the course, students were not always inclined to write tests. The instantaneous feedback of the automatic specification tests did not help in this respect. This effect has also been observed by earlier research [1]. In the 2015–2016 edition we introduced separate assignments to write tests for a language. To support this, automatically testing and grading for these tests was required.

Meta-Test Procedure We developed the meta-test suite outlined in Figure 17. It runs a student test suite against a collection of faulty interpreters, applying the good test criterion: *A good test is a test that fails if and only if a flaw*

```
abstract class TestSpecBase extends FunSuite {
def getTests: List[Group]
def generate(): String = {
  val groups = this.getTests.map(groupToString).mkString

  s"""|class $testName extends FunSuite
      |    with BeforeAndAfterAll {
      |$groups
      |}
      |""".stripMargin
}

def groupToString(group: Group): String =
  group match { case Group(_, _, clusters) =>
    clusters.map(clusterToString(group,
      sumOfClusterWeights, _)).mkString
}

def clusterToString(group: Group,
      sumOfClusterWeights: Double,
      cluster: Cluster): String = {
  ensureNotAllNegative(group, cluster)

  cluster.tests.map(test => generateTestCase(group,
    cluster, test)).mkString("\n")
}

def generateTestCase(group: Group, cluster: Cluster,
    test: Test): String = test match {
  case Pos(input, env) =>
    val either: Either[Throwable, List[Product]] = for {
      p <- tryParse(input).right
      d <- tryDesugar(p).right
      i <- tryInterp(d).right
    } yield List(p, d, i)

    either match {
      case Left(ex) =>
        throw new PosTestFailedException()
      case Right(pp :: dp :: ip :: Nil) =>
        this.emitParseAssert(input, pp)
        + this.emitDesugar(pp)
        + this.emitInterpAssert(dp, ip)
    }
  case Neg(input, env) => /* ... */
}
}
```

Figure 14. Test generator.

is introduced in the system under test. That is, testing a test boils down to supplying the test an implementation with a flaw and verifying that the test fails. Conversely, the test should not fail when the provided implementation does not contain any flaws. Students write one full test suite which means that the full test suite must be evaluated to verify that even a single error in the implementation is caught.

The `injectInterp(i: Interp): FunSuite` method finds the student test suite on the class path and then replaces its default interpreter field for the (faulty) interpreter provided. The modified student suite is then returned so the tests can be run with our custom runner.

```
// this will succeed if the x in the outer application
// is evaluated in the environment of the closure
Neg("((let ((x 1)) (lambda () x)) x)"),
```

Figure 15. Specification of complex test case in embedded test DSL.

```
// NEG: ((let ((x 1)) (lambda () x)) x)
assertResult(AppExt(LetExt(List(LetBindExt("x",
    NumExt(1))), FdExt(List(), IdExt("x"))),
    List(IdExt("x")))){
  Parser.parse("""((let ((x 1)) (lambda () x)) x)""")
}
Desugar.desugar(AppExt(LetExt(List(LetBindExt("x",
    NumExt(1))), FdExt(List(), IdExt("x"))),
    List(IdExt("x"))))
intercept[InterpException] {
  Interp.interp(AppC(AppC(FdC(List("x"), FdC(List(),
      IdC("x"))), List(NumC(1))), List(IdC("x"))))
}
```

Figure 16. Code generated from test case in Figure 15.

Defining Faulty Interpreters Following the definition of a good test, we need a set of faulty interpreters that each introduce exactly one fault. To avoid a maintenance nightmare, the introduction of a fault should not require duplicating all code of an interpreter. To achieve this goal, we use inheritance with method overriding to efficiently inject faults in a reference solution.

For example, a correct reference solution for an interpreter contains a lookup method. This method takes the current variable scope, and finds the value for a bound variable. The method should return an `UnboundIdException` when the variable is not in scope as defined in Figure 18. To introduce a flaw in this interpreter, we override the lookup method and return `NumV(-1)` whenever an identifier is not in scope as defined in Figure 19. We then inject the extended `BasicInterp` into the student test suite.

In order to allow this style of fault injection, we refactored the reference implementations of interpreters in order to expose bodies of cases in the interpretation function with calls to separate semantic functions.

Preventing Test Tampering The algorithm has several ways to detect test suites that are created to cheat the system. First of all a test suite with just one test which always fails would score all points. Therefore before the actual `MetaTest` is run, in the `beforeAll` function of the `MetaTest` suite it is verified that the student test suite passes on a correct implementation. While this does solve the issue for always-failing tests, a simple test suite that only succeeds once and then always fails would still be awarded the full score. The last cheat prevention mechanism employs running the test suite a random number of times to prevent students from hardcoding test based on the number of test invocations.

```
trait AbsMetaTest extends FunSuite
    with BeforeAndAfterAll {

  /* Prevent students from checking certain
   counts and succeed/fail based on the count */
  var max = Math.abs(new Random().nextInt() \% 10)
  for (a <- 1 to max) {
    testInterp(createInterp(), expectFail = false)
  }
  override def beforeAll() = {
    testInterp(createInterp(), expectFail = false)
  }

  def testInterp(i: Interp, expectFail: Boolean=true) =
    testSuite(injectInterp(i), expectFail)

  /* Implement per week */
  protected def injectInterp(i: Interp): FunSuite

  protected def testSuite(suite: FunSuite,
      expectFail: Boolean) = {
    var failed = false
    // Reporter used to verify at least one test failed
    val reporter: Reporter = new Reporter {
      override def apply(event: Event) = {
        event match {
          case _: TestFailed => failed = true
          case e: TestSucceeded => reportSuccess(e)
          case _ => // Do nothing
        }
      }
    }
    suite.run(None, reporter, new Stopper,
        new Filter(None, Set()), Map[String, Any](),
        None, new Tracker)

    // Assert that the suite correctly failed or passed
    //  on the given implementation.
    assert(failed == expectFail)
  }
}
```

Figure 17. Meta-test procedure for testing test suites.

7. Product Line

During the course, students are tasked with building an interpreter each week for a language with specific features. For each week in the course one reference interpreter is used to automatically generate output for specification tests. In the first version of the course these interpreters were all completely stand-alone Scala classes. Each interpreter was isolated from the others by a package structure. The interpreters had a high amount of overlap in functionality, because many language features were reused in multiple weeks of the course. This approach quickly proved to be hard to maintain as any change in the basic functionality had to be applied to each interpreter. Therefore we developed a solution to re-use code shared between interpreters.

Towards reuse The assignments for each week build on solutions of previous weeks. Sometimes, an assignment requires partial solutions of multiple previous assignments. The goal was to ease the burden of maintaining separate

```
class BasicInterp {
  // ... other implementation details omitted

  def lookup(name: String, nv: List[Bind]): Value =
    nv.find(x => x.name == name)
      .getOrElse(throw UnboundIdException(name)).value
}
```

Figure 18. Part of basic interpreter.

```
test("Lookup does not throw exception") {
  testInterp(new BasicInterp {
    override def lookup(name: String, env: List[Bind]) =
      env.find(x => x.name == name)
         .getOrElse(NumV(-1)).value
  })
}
```

Figure 19. Fault introduction in basic interpreter.

interpreters while allowing instructors to easily reconfigure assignments. An instructor determining the course schedule must be able to combine language features into assignments according to the desired schedule. More technically speaking, the goal was to have one stand-alone interpreter for each desired language feature as opposed to one interpreter for each week in the course. Ideally, the reference solution for each week should be a simple composition of the required language features.

Traits and multiple inheritance After several iterations, multiple inheritance using traits proved to be the best solution. The solution makes composing interpreters as simple as creating a trait mixin:

```
class ParserWeek5 extends BaseParser with TypedParser
    with ListParser with ParserWeek3
```

The above statement defines the reference parser used in week 5. This same construct is used for composing desugar- and interpreter-traits.

Each trait defines a method that parses an s-expression into a high-level AST (that includes sugar) of the target language: `parse(expr: SExpr): ExprExt`. Standard pattern matching in Scala is used to unmarshall the `SExpr` case class into an AST. When a trait fails to match a pattern, the fall-through case will delegate parsing to its parent in the inheritance linearization: `case _ => super.parse(sexpr)`. By making the call to `super`, the next trait in the inheritance linearization will be called [8]. When no trait can match the given expression, the fall through case will call `BaseParser` which throws an exception. Moreover, when a pattern matches and a recursive step is required, `this.parse(...)` is called. This call will be dispatched to the first trait in the linearization.

Conclusion This approach provides easy composition of language features. It allows instructors to structure a weekly assignment with a high amount of freedom. Since interpreters are separated by features rather than by course schedule, the maintainability of multiple interpreters is greatly improved. One disadvantage of this approach is the verbosity of the trait mix-in classes for each week. However, the impact of this verbosity on the overall maintainability is very low. Another disadvantage is that language features can only be composed if the types of their ASTs, parsers and interpreters are compatible. For example, due to the introduction of mutation, the method signature of the interpreter changes. This mismatch in method signatures makes it impossible to create a trait mixin with other interpreters.

The overall problem of extensibility is also known as the expression problem [9, 16]. Solutions to this problem involve trait extensions of a base class which processes one expression at a time. Our solution differs in the sense that one processor can handle multiple expressions. There is no guarantee that a certain expression is processed by a composition of traits of sub-processors. This type safety is not a problem for us since we have a known set of expressions to be processed. We make use of `super` calls to delegate processors while we manually make sure every expression is processed by at least one mixed in trait.

8. Conclusion

In this paper we have discussed how, through leveraging the Scala programming language and the WebLab online learning management system to automatically run specification tests on the students submissions, we have developed a scalable solution for running a course on concepts of programming languages using definitional interpreters.

Scala has proven to be a convenient language for this purpose. It supports a functional style of programming with algebraic data types and pattern matching, which is suitable for implementation of big-step interpreters. At the same time it provides a well developed programming language and ecosystem allowing us to develop the course as a collection of reusable components. This includes a malleable unit testing framework that could be used to test student tests suites, express an internal DSL for test generation, and develop a product line of interpreters.

As part of future work, we are planning to investigate techniques to provide feedback to students on other aspects than functional correctness. This feedback will involve static analysis of the code written by a student. For example check that the solution of the student has methods no longer than a number of lines.

References

[1] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *ACM Journal of Educational Resources in Computing*, 3(3):1–24, 2003.

[2] S. N. Kamin. *Programming languages - an interpreter-based approach*. Addison-Wesley, 1990.

[3] L. C. L. Kats, R. Vogelij, K. T. Kalleberg, and E. Visser. Software development environments on the web: a research agenda. In G. T. Leavens and J. Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, pages 99–116. ACM, 2012.

[4] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2007.

[5] S. Krishnamurthi and J. G. Politz. *Programming and Programming Languages*. Brown University, 2015.

[6] H. Miller, P. Haller, L. Rytz, and M. Odersky. Functional programming for all! scaling a mooc for students and professionals alike. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 256–263. ACM, 2014.

[7] H. R. Nielson and F. Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[8] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima, November 2008.

[9] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proceedings of the Twelth InternationalWorkshop on Foundations of Object-Oriented Languages (FOOL 12)*, 2005.

[10] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, B. Yorgey, B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. 2015.

[11] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[12] J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[13] R. Sethi. *Programming languages - concepts and constructs*. Addison-Wesley, 1988.

[14] B. Venners. Scalatest, 2009.

[15] V. Vergu. LabBack: An extendible platform for secure and robust in-the-cloud automatic assessment of student programs. Master's thesis, Delft University of Technology, November 2012.

[16] P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[17] D. A. Watt and W. Findlay. *Programming language design concepts*. Wiley, 2004.