Type Systems

Martin Odersky
EPFL

Master level course, 2006/2007

October 25, 2006 - version 1.1

Slides in part adapted from:
University of Pennsylvania CIS 500: Software Foundations - Fall 2006
by Benjamin Pierce

Course Overview

The main part of this course is about type systems

But in a larger sense the course is about software foundations

What is "software foundations"?

Software foundations (or "theory of programming languages") is the mathematical study of the meaning of programs.

The goal is finding ways to describe program behaviors that are both precise and abstract.

- precise so that we can use mathematical tools to formalize and check interesting properties
- abstract so that properties of interest can be discussed clearly, without getting bogged down in low-level details

Why study software foundations?

- To prove specific properties of particular programs (i.e., program verification)
  - Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for *informal* reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To understand language features (and their interactions) deeply and develop principles for better language design
  (PL is the "materials science" of computer science...)

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

Most software designers are language designers!

What this course is not

- An introduction to programming
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- A comparative survey of many different programming languages and styles

Approaches to Program Meaning

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.
- Type systems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

In this course, we will concentrate on operational techniques and type systems.

- Part I: Modeling programming languages
  - Syntax and parsing
  - Operational semantics
  - Inductive proof techniques
  - The lambda-calculus
  - Syntactic sugar; fully abstract translations
- Part II: Type systems
  - Simple types
  - Type safety
  - References
  - Subtyping

Overview

- Part III: Object-oriented features (case study)
  - A simple imperative object model
  - An analysis of core Java
  - An analysis of core Scala

Organization of the Course

People

Instructor: Martin Odersky
INR 319
<martin.odersky@epfl.ch>

Co-instructors: Lex Spoon
<lex@lexspoon.org>
Sean McDirmid
<sean.mcdirmid@epfl.ch>

Teaching Assistants: Iulian Dragos
<iulian.dragos@epfl.ch>
Stephane Micheloud
<stephane.micheloud@epfl.ch>

Information

Textbook: Types and Programming Languages,
Benjamin C. Pierce, MIT Press, 2002

Webpage: `http://lampwww.epfl.ch/teaching/typeSystems/`

Elements of the Course

- The Type Systems course consists of
  - lecture (Wednesday 13:15-15:00, room INM 203)
  - exercises and project work (Wednesday 15:15-17:00, room INF 1)
- The lecture will follow in large parts the textbook.
- For lack of time, we cannot treat all essential parts of the book in the lectures, that's why the textbook is required reading for participants of the course.

Homework and Projects

You will be asked to

- solve and hand in some written exercise sheets,
- do a number of programming assignments, including
  - parsers,
  - interpreters and reduction engines,
  - type checkers

  for a variety of small languages.
- The recommended implementation language for these assignments is Scala.

Scala

- Scala is a functional and object-oriented language that is closely interoperable with Java.
- It is very well suited as an implementation language for type-checkers, in particular because it supports:
  - pattern matching,
  - higher-order functions,
  - inheritance and mixins.

Learning Scala

If you don't know Scala yet, there's help:

- The Scala web site:

  scala.epfl.ch

- On this site, the documents:
  - *A Brief Scala Tutorial - an introduction to Scala for Java programmers.* (short and basic).
  - *An Introduction to Scala* (longer and more comprehensive).
  - *An Overview of the Scala Programming Language* (high-level).
  - *Scala By Example* (long, comprehensive, tutorial style).
- The assistants.

Grading and Exams

Final course grades will be computed as follows:

- Homework and project: 30%
- Mid-term exam: 30% each
- Final exam: 40%

Exams:

1. Mid-term: Wed, Dec 20th, 2006
2. Final exam: Wed, Feb 7th, 2007

(dates are provisional)

Collaboration

- Collaboration on homework is strongly encouraged.
- Studying with other people is the best way to internalize the material
- Form pair programming and study groups!
  2-3 people is a good size. 4 is too many for all to have equal input.

"You never really misunderstand something
until you try to teach it...
" – Anon.

Part I

Modelling programming languages

Syntax and Parsing

- The first-level of modeling a programming language concerns its context-free syntax.
- Context free syntax determines a set of legal phrases and determines the (tree-)structure of each of them.
- It is often given on two levels:
  - concrete: determines the exact (character-by-character) set of legal phrases
  - abstract: concentrates on the tree-structure of legal phrases.
- We will be mostly concerned with abstract syntax in this course.
- But to be able to write complete programming tools, we need a convenient way to map character sequences to trees.

Approaches to Parsing

There are two ways to construct a parser:

- By hand Derive a parser program from a grammar.
- Automatic Submit a grammar to a tool which generates the parser program.

In the second approach, one uses a special grammar description language to describe the input grammar.

Domain-Specific Languages

- The grammar description language is an example of a domain-specific language (DSL).
- The parser generator acts as a processor ("compiler") for this language — that's why it's sometimes called grandly a "compiler-compiler".
- Example of a "program" in the grammar description DSL:

```
Expr   ::= Term {'+' Term | '-' Term}.
Term   ::= Factor {'*' Factor | '/' Factor}.
Factor ::= Number | '(' Expr ')'.
```


Hosted Domain Specific Languages

- An alternative to a stand-alone DSL is a hosted DSL.
- Here, the DSL does not exist as a separate language but as an API in a host language.
- The host language is usually a general purpose programming language.

We will now develop this approach for grammar description languages.

A Hosted Grammar Description Language in Scala

We will develop a framework where grammars can be described like this:

```
def Expr   : Parser = Term & rep(kw("+") & Term | kw("−") & Term)
def Term   : Parser = Factor & rep(kw("∗") & Factor | kw("/") & Factor)
def Factor : Parser = numericLit | kw("(") & Expr & kw(")")
```

This description can be produced from the previous grammar by systematic text replacements:

- Insert a def at the beginning of each production.
- The "::=" becomes ": Parser =".
- Sequential composition is now expressed by a &.
- Repetition {...} is now expressed by rep(...).
- Option [...] is now expressed by opt(...).

- Terminal symbols appear inside kw(...).
- The point at the end of a production is removed.

Parser Combinators

- The differences between Grammar A and Grammar B are fairly minor.

  (Note in particular that existing DSL's for grammar descriptions also tend to add syntactic complications to the idealized Grammar A we have seen).

- The important difference is that Grammar B is a valid Scala program, when combined with an API that defines the necessary primitives.
- These primitives are called parser combinators.

The Basic Idea

For each language (identified by grammar symbol S), define a function f_S that, given an input stream i,

- if a prefix of i is in S, return Success(Pair(x, i')) where x is a result for S and i' is the rest of the input.
- otherwise, return Failure(msg, i) where msg is an error message string.

The first behavior is called success, the second failure.

The Basic Idea in Code

```
class GenericParsers {
    type Parser = Input => ParseResult
```

where

```
    type Input = List[Token]   or   type Input = Stream[Token]
```

and we assume:

- A class Token with subclasses
  - case class KW(chars : String) for keywords,
  - case class NumericLit(chars : String) for numbers,
  - case class StringLit(chars : String) for strings,
  - case class Identifier(chars : String) for identifiers.

  In each case, chars represents the characters making up the token.

- A class ParseResult with subclasses

```
case class Success(out : List[Token], in : Input)
extends ParseResult

case class Failure(msg : String, in : Input)
extends ParseResult
```

Object-Oriented Parser Combinators

- In fact, we will also need to express | and & as methods of parsers.
- That's why we extend the function type of parsers as follows:

```
abstract class Parser extends (Input => ParseResult) {
   // An unspecified method that defines the parser function.
   def apply(in: Input): ParseResult

   // A parser combinator for sequential composition
   def & ...

   // A parser combinator for alternative composition
   def | ...
}
```

It remains to define concrete combinators that implement this class (see below).

The Generic Single-Token Parser

- The following parser succeeds if the first token in the input satisfies a given predicate p.
- If it succeeds, it reads the token and returns it as a result.

```
def token(kind: String, p: Token => boolean) = new Parser {
   def apply(in: Input) =
      if (p(in.head)) Success(List(in.head), in.tail)
      else Failure(kind+" expected", in)
}
```

Specific Single-Token Parsers

- The following parser succeeds if the first token in the input is a given keyword "chars":
- If it succeeds, it returns a keyword token as a result.

```
def kw(chars: String) = token("'"+chars+"'", {
  case KW(chars1) => chars == chars1
  case _ => false
})
```

- The following parsers succeed if, respectively, the first token in the input is a numeric or string literal, or an identifier.

```
def numericLit = token("number", .isInstanceOf[NumericLit])
def stringLit = token("string literal", .isInstanceOf[StringLit])
def ident = token("identifier", .isInstanceOf[Identifier])
```


The Sequence Combinator

- The sequence combinator P & Q succeeds if P and Q both succeed. It then returns a list whose containing the concatenation of result of P and the result of Q.
- & is implemented as a method of class Parser.

```
abstract class Parser {
  private var p = this
  def & (q: Parser) = new Parser {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) => Success(x ::: y, in2)
          case failure => failure
        }
      case failure => failure
    }
  }
```

The Alternative Combinator

- The alternative combinator $P \mid Q$ succeeds if either P or Q succeeds.
- It returns the result of P if P succeeds, or the result of Q, if Q succeeds.
- The alternative combinator is implemented as a method of class Parser.

```
def | (q : => Parser) = new Parser {
  def apply(in : Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case f1 @ Failure(_, in1) => q(in) match {
      case s2 @ Success(_, _) => s2
      case f2 @ Failure(_, in2) =>
        if (in2.head.pos < in1.head.pos) f1 else f2
    }
}}
```


Failure And Success Parsers

- The parser failure(msg) always fails with the given error message. It is implemented as follows:

```
def failure(msg : String) = new Parser {
  def apply(in : Input) = Failure(msg, in)
}
```

- The parser success(result) always succeeds with the given result. It does not consume any input. It is implemented as follows:

```
def success(result : List[Token]) = new Parser {
  def apply(in : Input) = Success(result, in)
}
```

- The parser empty always succeeds with the empty list as result:

```
def empty = success(List())
```

Option and Repetition Combinators

- The opt(P) combinator is equivalent to P if P succeeds and is equivalent to empty if P fails.
- The rep(P) combinator applies P zero or more times until P fails.

The two combinators are implemented as follows:

```
def opt(p : Parser): Parser  = p | empty
def rep(p : Parser): Parser  = p & rep(p) | empty
```

Note that neither of these combinators can fail!

Other Combinators

More combinators can be defined if necessary.

Exercise: Implement the rep1(P) parser combinator, which applies P one or more times.

Exercise: Define opt and rep directly, without making use of &, |, and empty.

Parser Output

- So far, the output of a parser was equal the list of input tokens.
- To get other kinds of output, we create a trait of BuildingParsers as a refinement of GenericParsers.
- These parsers introduce a new class of token, the nonterminal symbol:

```
trait BuildingParsers extends GenericParsers {
   type Output
   case class NonTerminal(value: Output)(pos: Position)
       extends Token(pos)
```

- A non-terminal symbol contains an output value (this is often, but not always, an abstract syntax tree).

Creating Non-Terminals

- Non-terminal symbols are produced by the production(P) parser combinator.
- This parser succeeds iff P succeeds. It returns a single non-terminal symbol.
- The value part of this symbol is constructed by calling method build with the list of tokens returned by P as parameter.
- The build method is abstract in GenericParsers; it needs to be defined by every concrete parser implementation. Its signature is as follows:

```
def build(elems: List[Token], pos: Position): Output
```

- Here is the implementation of the production parser:

```
def production(p: Parser) = new Parser {
  def apply(in: Input) = p(in) match {
    case Success(out, in1) ⇒
      Success(
        List(
          NonTerminal(build(out, in.head.pos))(in.head.pos)),
          in1)
    case failure ⇒ failure
  }
}
```


Example: Evaluating Arithmetic Expression

We now design a program that

- accepts as input an arithmetic expression (given as a string)
- yields as output the result of evaluating the expression,
- or produces an error message if the expression is not well-formed.

The program uses the combinator parsing library:

```
object ExprParser extends TokenizingParsers // it parses from array of characters
                  with BuildingParsers      // it constructs some result
                  with Application {        // it has a main method
  ...
```

Lexical Analysis

- Class TokenizingParsers contains a method

```
def tokenize(source: Array[char]): Iterator[Token]
```

that yields a sequence of tokens from an array of characters, source.

- It is driven by two sets:

```
// The delimiters used for the tokenizer
// A delimiter is a symbol which always constitutes a single character
// token, even if not preceded or followed by whitespace.
protected val delimiters = new HashSet[char]

// The keywords used for the tokenizer
// Strings matching a keyword yield KW tokens instead of
// Identifier tokens. Delimiter tokens are always keywords,
// no need to add their names to the 'keywords' set.
protected val keywords = new HashSet[String]
```


Evaluating arithmetic expression ctd.

- Our expressions have the following delimiters:

```
delimiters.incl('(', '+', '−', '/', '∗', ')')
```

- They need no other keywords.

Arithmetic expression grammar

- The rest of the grammar is as before:

```
def Expr : Parser = production (
    Term & rep(kw("+") & Term | kw("−") & Term)
)
def Term : Parser = production (
    Factor & rep(kw("∗") & Factor | kw("/") & Factor)
)
def Factor : Parser = production (
     numericLit
    | kw("(") & Expr & kw(")")
    | failure("illegal start of expression")
)
```


Computing Results

- To compute expression results, we first have to fix the Output type and the build function:

```
type Output = int
def build(xs : List[Token], pos : Position) = eval(xs)
```

- We then have to write an evaluation function, which takes a list of tokens and yields an integer result.

```
def eval(xs : List[Token]): Output = xs match {
    case NonTerminal(t1) :: KW("+") :: NonTerminal(t2) :: rest ⇒
        eval(NonTerminal(t1 + t2) :: rest)
    // analogous for -, *, and /
    case List(NumericLit(n)) ⇒
        Integer.parseInt(n)
    case List(KW("("), t, KW(")")) ⇒
        t
    case List(NonTerminal(result)) ⇒
        result }
```

A bit of polishing

- The wrapping and unwrapping of results into/from non-terminals is a bit bulky.
- We can hide it by using two implicit conversions which are defined in BuildingParsers.
-

```
protected object implicitConversions {
  implicit def token2output(n : Token): Output =
    n.asInstanceOf[NonTerminal].value
  implicit def output2token(n : Output): Token =
    NonTerminal(n)(NoPosition)
}
```

- These implicit conversion get applied automatically when they are visible in some scope.

Computing Results (2)

- Here is the evaluation function, with implicit conversions enabled.

```
import implicitConversions._ // enable implicit conversions

def eval(xs : List[Token]): Output = xs match {
  case t1 :: KW("+") :: t2 :: rest ⇒
    eval((t1 + t2) :: rest)
  // analogous for -, *, and /
  case List(NumericLit(n)) ⇒
    Integer.parseInt(n)
  case List(KW("("), t, KW(")")) ⇒
    t
  case List(NonTerminal(result)) ⇒
    result
}
```