Elements of Programming Languages

Lecture 0: Introduction and Course Outline

James Cheney

University of Edinburgh

September 20, 2016

What is programming?

- Computers are deterministic machines, controlled by low-level (usually binary) machine code instructions.
- A computer **can [only] do whatever we know how to order it to perform** (Ada Lovelace, 1842)
- Programming is **communication**:
  - between a person and a machine, to tell the machine what to do
  - between people, to communicate ideas about algorithms and computation

From machine code to programming languages

- The first programmers wrote all of their code directly in machine instructions
  - ultimately, these are just raw sequences of bits.
- Such programs are extremely difficult to write, debug or understand.
- Simple "assembly languages" were introduced very early (1950's) as a human-readable notation for machine code
- FORTRAN (1957) — one of the first "high-level" languages (procedures, loops, etc.)

What is a programming language?

- For the purpose of this course, a programming language is a *formal*, *executable* language for *computations*
- Non-examples:

What is a programming language?

- For the purpose of this course, a programming language is a *formal*, *executable* language for *computations*
- Non-examples:
  - English (not formal)

Introduction Course Administration Course Outline

What is a programming language?

- For the purpose of this course, a programming language is a *formal*, *executable* language for *computations*
- Non-examples:
  - English (not formal)
  - First-order Logic (formal, but not executable in general)

Introduction Course Administration Course Outline

What is a programming language?

- For the purpose of this course, a programming language is a *formal*, *executable* language for *computations*
- Non-examples:
  - English (not formal)
  - First-order Logic (formal, but not executable in general)
  - HTML4 (formal, executable but not computational)

Introduction Course Administration Course Outline

What is a programming language?

- For the purpose of this course, a programming language is a *formal*, *executable* language for *computations*
- Non-examples:
  - English (not formal)
  - First-order Logic (formal, but not executable in general)
  - HTML4 (formal, executable but not computational)
- (HTML is in a gray area — with JavaScript or HTML5 extensions it is a lot more "computational")

Why are there so many?

- Imperative/procedural: FORTRAN, COBOL, Algol, Pascal, C
- Object-oriented, untyped: Simula, Smalltalk, Python, Ruby, JavaScript
- Object-oriented, typed: C++, Java, Scala, C#
- Functional, untyped: LISP, Scheme, Racket
- Functional, typed: ML, OCaml, Haskell, (Scala), F#
- Logic/declarative: Prolog, Curry, SQL

What do they have in common?

- All (formal) languages have a written form: we call this (concrete) *syntax*
- All (executable) languages can be implemented on computers: e.g. by a *compiler* or *interpreter*
- All programming languages describe computations: they have some *computational meaning*, or*semantics*
- In addition, most languages provide *abstractions* for organizing, decomposing and combining parts of programs to solve larger problems.

What are the differences?

There are many so-called "programming language paradigms":

- imperative (variables, assignment, if/while/for, procedures)
- object-oriented (classes, inheritance, interfaces, subtyping)
- typed (statically, dynamically, strongly, un/uni-typed)
- functional (λ-calculus, pure, lazy)
- logic/declarative (computation as deduction, query languages)

Languages, paradigms and elements

- A great deal of effort has been expended trying to find the "best" paradigm, with no winner declared so far.
- In reality, they all have strengths and weaknesses, and almost all languages make compromises or synthesize ideas from several "paradigms".
- This course emphasizes different programming language **features**, or **elements**
  - Analogy: periodic table of the elements in chemistry
- Goal: understand the basic components that appear in a variety of languages, and how they "combine" or "react" with one another.

Applicability

- Major new general-purpose languages come along every decade or so.
  - Hence, few programmers or computer scientists will design a new, widely-used general purpose language, or write a compiler
  - However, domain-specific languages are increasingly used, and the same principles of design apply to them
- Moreover, understanding the principles of language design can help you become a better programmer
  - Learn new languages / recognize new features faster
  - Understand when and when *not* to use a given feature
- Assignments will cover practical aspects of programming languages: *interpreters* and *DSLs/translators*

Course Administration

Staff

- Lecturer: James Cheney <jcheney@inf.ed.ac.uk>, IF 5.29
  - Office hours: Monday 11:30-12:30, or by appointment
- TA: TBA

Format

- 20 **lectures** (Tu/F 1410–1500)
  - 2 intro/review [non-examinable]
  - 2 guest lectures [non-examinable]
  - 16 core material [examinable]
- 1 two-hour **lab session** (September 28, 1210–1400)
- 8 one-hour **tutorial sessions**, starting in week 3 (times and groups TBA)

All of these activities are **part of the course** and may cover examinable material, unless explicitly indicated.

Feedback and Assessment

- Coursework:
  - Assignment 1: **Lab exercise sheet**, available during week 2, due during week 3, worth 0% of final grade
  - Assignment 2: available during week 3, due week 6, worth 0% of final grade.
  - Assignment 3: available during week 6, due week 10, worth 25% of final grade.
  - The first two assignments are marked for formative feedback only, but the third **builds on the first two**.
- One (written) exam: worth 75% of final grade.

Scala

- The main language for this course will be *Scala*
  - Scala offers an interesting combination of ideas from functional and object-oriented programming styles
  - We will use Scala (and other languages) to illustrate key ideas
  - We will also use Scala for the assignments
- However, this is not a "course on Scala"
  - You will be expected to figure out certain things for yourselves (or ask for help)
  - We will not teach every feature of Scala, nor are you expected to learn every dark corner
  - In fact, part of the purpose of the course is to help you recognize such dark corners and avoid them unless you have a good reason...

Recommended reading

- There is no official textbook for the course that we will follow exactly
- However, the following are recommended readings to complement the course material:
  - Practical Foundations for Programming Languages, second edition, (PFPL2), by Robert Harper. Available online from the author's webpage and through the University Library's ebook access.
  - Concepts in Programming Languages (CPL), by John Mitchell. Available through the University Library's ebook access.
- The webpage lecture outline will indicate relevant sections and additional suggested readings

Course Outline

Wadler's Law

In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics
1. Syntax
2. Lexical syntax
3. Lexical syntax of comments

Wadler's law is an example of a phenomenon called "bike-shedding":

- the number of people who feel qualified to comment on an issue is inversely proportional to the expertise required to understand it

Syntax

- This course is primarily about language design and semantics.
- As a foundation for this, we will necessarily spend some time on abstract syntax trees (and programming with them in Scala)
- We will cover: Name-binding, substitution, static vs. dynamic scope
- We will not cover: Concrete syntax, lexing, parsing, precedence (but Compiling Techniques does)

Interpreters, Compilers and Virtual Machines

- Suppose we have a *source* programming language L_S, a *target* language L_T, and an *implementation* language L_I
  - An *interpreter* for L_S is an L_I program that executes L_S programs.
  - When both L_S and L_I are low-level (e.g. L_S = JVM, L_I = x86), an interpreter for L is called a *virtual machine*.
  - A *translator* from L_S to L_T is an L_I program that translates programs in L_S to "equivalent" programs in L_T.
  - When L_T is low-level, a translator to L_T is usually called a *compiler*.
- In this course, we will use interpreters to explore different language features.

Semantics

- How can we understand the meaning of a language/feature, or compare different languages/features?
- Three basic approaches:
  - *Operational semantics* defines the meaning of a program in terms of "rules" that explain the step-by-step execution of the program
  - *Denotational semantics* defines the meaning of a program by interpreting it in a mathematical structure
  - *Axiomatic semantics* defines the meaning of a program via logical specifications and laws
- All three have strengths and weaknesses
- We will focus on operational semantics in this course: it is the most accessible and flexible approach.

The three most important things

- The three most important considerations for programming language design are:
  - (Data) Abstraction
  - (Control) Abstraction
  - (Modular) Abstraction
- We will investigate different language elements that address the need for these abstractions, and how different design choices interact.
- In particular, we will see how **types** offer a fundamental organizing principle for programming language features.

Data Structures and Abstractions

- **Data structures** provide ways of organizing data:
  - option types vs. null values
  - pairs/record types;
  - variant/union types;
  - lists/recursive types;
  - pointers/references
- **Data abstractions** make it possible to hide data structure choices:
  - overloading (ad hoc polymorphism)
  - generics (parametric polymorphism)
  - subtyping
  - abstract data types

Control Structures and Abstractions

- **Control structures** allow us to express flow of control:
  - goto
  - for/while loops
  - case/switch
  - exceptions
- **Control abstractions** make it possible to hide implementation details:
  - procedure call/return
  - function types/higher-order functions
  - continuations

Design dimensions and modularity

- Programming "in the large" requires considering several cross-cutting **design dimensions**:
  - eager vs. lazy evaluation
  - purity vs. side-effects
  - static vs. dynamic typing
- and **modularity** features
  - modules, namespaces
  - objects, classes, inheritance
  - interfaces, information hiding

The art and science of language design

- Language design is both an art and a science
- The most popular languages are often not the ones with the cleanest foundations (and vice versa)
- This course teaches the science: formalisms and semantics
- Aesthetics and "good design" are hard to teach (and hard to assess), but one of the assignments will give you an opportunity to experiment with domain-specific language design

Course goals

By the end of this course, you should be able to:

1. Investigate the design and behaviour of programming languages by studying implementations in an interpreter
2. Employ abstract syntax and inference rules to understand and compare programming language features
3. Design and implement a domain-specific language capturing a problem domain
4. Understand the design space of programming languages, including common elements of current languages and how they are combined to construct language designs
5. Critically evaluate the programming languages in current use, acquire and use language features quickly, recognise problematic programming language features, and avoid their (mis)use.

Relationship to other courses

- **Compiling Techniques**
  - covers complementary aspects of PL implementation, such as lexical analysis and parsing.
  - also covers compilation of imperative programs to machine code
- **Introduction to Theoretical Computer Science**
  - covers formal models of computation (Turing machines, etc.)
  - as well as some λ-calculus and type theory
- In this course, we focus on *interpreters*, *operational semantics*, and *types* to understand programming language features.
- There should be relatively little overlap with CT or ITCS.

Summary

- Today we covered:
  - Background and motivation for the course
  - Course administration
  - Outline of course topics
- Next time:
  - Concrete and abstract syntax
  - Programming with abstract syntax trees (ASTs)

Elements of Programming Languages

Lecture 1: Abstract syntax

James Cheney

University of Edinburgh

September 23, 2016

Today

We will introduce some basic tools used throughout the course:

- Concrete vs. abstract syntax
- Abstract syntax trees
- Induction over expressions

L_{Arith}

- We will start out with a very simple (almost trivial) "programming language" called L_{Arith} to illustrate these concepts
- Namely, expressions with integers, $+$ and $\times$
- Examples:

```
1 + 2         ---> 3
1 + 2 * 3     ---> 7
(1 + 2) * 3 ---> 9
```


Concrete vs. abstract syntax

- **Concrete syntax:** the actual syntax of a programming language
  - Specify using context-free grammars (or generalizations)
  - Used in compiler/interpreter front-end, to decide how to interpret **strings** as programs
- **Abstract syntax:** the "essential" constructs of a programming language
  - Specify using so-called *Backus Naur Form* (BNF) grammars
  - Used in specifications and implementations to describe the *abstract syntax trees* of a language.

CFG vs. BNF

- Context-free grammar giving concrete syntax for expressions

$$
\begin{array}{lcl}
E & \rightarrow & E \; \texttt{PLUS} \; F \mid F \\
F & \rightarrow & F \; \texttt{TIMES} \; F \mid \texttt{NUM} \mid \texttt{LPAREN} \; E \; \texttt{RPAREN}
\end{array}
$$

- Needs to handle precedence, parentheses, etc.
- Tokenization ($+ \rightarrow$ PLUS, etc.), comments, whitespace usually handled by a separate stage

BNF grammars

- BNF grammar giving abstract syntax for expressions

$$Expr \ni e \quad ::= \quad e_1 + e_2 \mid e_1 \times e_2 \mid n \in \mathbb{N}$$

- This says: there are three kinds of expressions
  - Additions $e_1 + e_2$, where two expressions are combined with the $+$ operator
  - Multiplications $e_1 \times e_2$, where two expressions are combined with the $\times$ operator
  - Numbers $n \in \mathbb{N}$
- Much like CFG rules, we can "derive" more complex expressions:

$$e \rightarrow e_1 + e_2 \rightarrow 3 + e_2 \rightarrow 3 + (e_3 \times e_4) \rightarrow \cdots$$

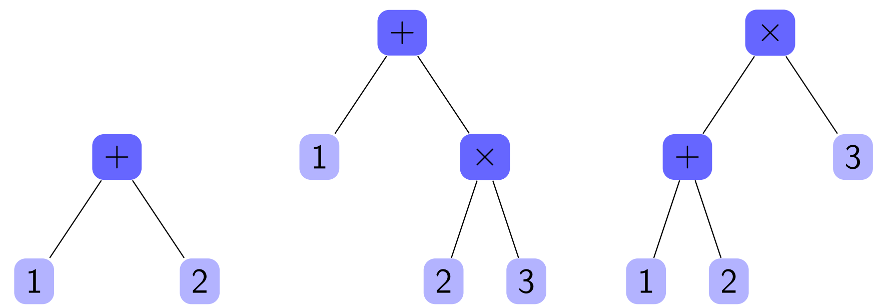
BNF conventions

- We will usually use BNF-style rules to define abstract syntax trees
  - and assume that concrete syntax issues such as precedence, parentheses, whitespace, etc. are handled elsewhere.
- **Convention:** the subscripts on occurrences of e on the RHS don't affect the meaning, just for readability
- **Convention:** we will freely use parentheses in abstract syntax notation to disambiguate
- e.g.

$$(1 + 2) \times 3 \qquad \text{vs.} \qquad 1 + (2 \times 3)$$

Abstract Syntax Trees (ASTs)

We view a BNF grammar to define a collection of *abstract syntax trees*, for example:



These can be represented in a program as trees, or in other ways (which we will cover in due course)

Languages for examples

- We will use several languages for examples throughout the course:
  - Java: typed, object-oriented
  - Python: untyped, object-oriented with some functional features
  - Haskell: typed, functional
  - Scala: typed, combines functional and OO features
  - Sometimes others, to discuss specific features
- You do not need to already know all these languages!

ASTs in Java

- In Java ASTs can be defined using a class hierarchy:

```
abstract class Expr {}
class Num extends Expr {
  public int n;
  Num(int _n) {
    n = _n;
  }
}
```


ASTs in Java

- In Java ASTs can be defined using a class hierarchy:

```
...
class Plus extends Expr {
  public Expr e1;
  public Expr e2;
  Plus(Expr _e1, Expr _e2) {
    e1 = _e1;
    e2 = _e2;
  }
}
class Times extends Expr {... // similar
}
```


ASTs in Java

- Traverse ASTs by adding a method to each class:

```
abstract class Expr {
  abstract public int size();
}
class Num extends Expr { ...
  public int size() { return 1;}
}
class Plus extends Expr { ...
  public int size() {
    return e1.size(e1) + e2.size() + 1;
  }
}
class Times extends Expr {... // similar
}
```

ASTs in Python

- Python is similar, but shorter (no types):

```
class Expr:
    pass # "abstract"
class Num(Expr):
    def __init__(self,n):
        self.n = n
    def size(self): return 1
class Plus(Expr):
    def __init__(self,e1,e2):
        self.e1 = e1
        self.e2 = e2
    def size(self):
        return self.e1.size() + self.e2.size() + 1
class Times(Expr): # similar...
```


ASTs in Haskell

- In Haskell, ASTs are easily defined as *datatypes*:

```
data Expr = Num Integer
          | Plus Expr Expr
          | Times Expr Expr
```

- Likewise one can easily write functions to traverse them:

```
size :: Expr -> Integer
size (Num n) = 1
size (Plus e1 e2) =
  (size e1) + (size e2) + 1
size (Times e1 e2) =
  (size e1) + (size e2) + 1
```


ASTs in Scala

- In Scala, can define ASTs conveniently using *case classes*:

```
abstract class Expr
case class Num(n: Integer) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
case class Times(e1: Expr, e2: Expr) extends Expr
```

- Again one can easily write functions to traverse them using pattern matching:

```
def size (e: Expr): Int = e match {
  case Num(n) => 1
  case Plus(e1,e2) =>
    size(e1) + size(e2) + 1
  case Times(e1,e2) =>
    size(e1) + size(e2) + 1
}
```


Creating ASTs

- Java:

```
new Plus(new Num(2), new Num(2))
```

- Python:

```
Plus(Num(2),Num(2))
```

- Haskell:

```
Plus(Num(2),Num(2))
```

- Scala: (the "new" is optional for case classes:)

```
new Plus(new Num(2),new Num(2))
Plus(Num(2),Num(2))
```

Precedence, Parentheses and Parsimony

- Infix notation and operator precedence rules are convenient for programmers (looks like familiar math) but complicate language front-end
- Some languages, notably LISP/Scheme/Racket, eschew infix notation.
- All programs are essentially so-called S-Expressions:

$$s ::= a \mid (a\ s_1\ \cdots\ s_n)$$

  so their concrete syntax is very close to abstract syntax.
- For example

```
1 + 2         ---> (+ 1 2)
1 + 2 * 3     ---> (+ 1 (* 2 3))
(1 + 2) * 3   ---> (* (+ 1 2) 3)
```


The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
  - (Structural) induction
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
    - (over $\mathbb{N}$)
  - (Structural) induction
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
    - (over $\mathbb{N}$)
  - (Structural) induction
    - (over ASTs)
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
    - (over $\mathbb{N}$)
  - (Structural) induction
    - (over ASTs)
  - (Rule) induction
    - (over derivations)
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

Induction

- Recall the *principle of mathematical induction*

Mathematical induction

Given a property P of natural numbers, if:

- $P(0)$ holds
- for any $n \in \mathbb{N}$, if $P(n)$ holds then $P(n+1)$ also holds

Then $P(n)$ holds for all $n \in \mathbb{N}$.

Induction over expressions

- A similar principle holds for expressions:

Induction on structure of expressions

Given a property P of expressions, if:

- $P(n)$ holds for every number $n \in \mathbb{N}$
- for any expressions e_1, e_2, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 + e_2)$ also holds
- for any expressions e_1, e_2, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 \times e_2)$ also holds

Then $P(e)$ holds for all expressions e.

- Note that we are performing induction over *abstract syntax trees*, not numbers!

Proof of expression induction principle

Define the *size* of an expression in the obvious way:

$$\begin{aligned} size(n) &= 1 \\ size(e_1 + e_2) &= size(e_1) + size(e_2) + 1 \\ size(e_1 \times e_2) &= size(e_1) + size(e_2) + 1 \end{aligned}$$

Given $P(-)$ satisfying the assumptions of expression induction, we prove the property

$$Q(n) = \text{for all } e \text{ with } size(e) < n \text{ we have } P(e)$$

Since any expression e has a finite size, $P(e)$ holds for any expression.

Proof of expression induction principle

Proof.

We prove that $Q(n)$ holds for all n by induction on n:

- The base case $n = 0$ is vacuous
- For $n + 1$, then assume $Q(n)$ holds and consider any e with $size(e) < n + 1$. Then there are three cases:
  - if $e = m \in \mathbb{N}$ then $P(e)$ holds by part 1 of expression induction principle
  - if $e = e_1 + e_2$ then $size(e_1) < size(e) \leq n$ and similarly for $size(e_2) < size(e) \leq n$. So, by induction, $P(e_1)$ and $P(e_2)$ hold, and by part 2 of expression induction principle $P(e)$ holds.
  - if $e = e_1 \times e_2$, the same reasoning applies.

$\square$

Concrete vs. abstract syntax Abstract syntax trees Structural Induction

Summary

- We covered:
  - Concrete vs. Abstract syntax
  - Abstract syntax trees
  - Abstract syntax of $\mathrm{L}_{\mathrm{Arith}}$ in several languages
  - Structural induction over syntax trees
- This might seem like a lot to absorb, but don't worry! We will revisit and reinforce these concepts throughout the course.
- Next time:
  - Evaluation
  - A simple interpreter
  - Operational semantics rules

Elements of Programming Languages

Lecture 2: Evaluation

James Cheney

University of Edinburgh

September 27, 2016

Overview

- Last time:
  - Concrete vs. abstract syntax
  - Programming with abstract syntax trees
  - A taste of induction over expressions
- Today:
  - Evaluation
  - A simple interpreter
  - Modeling evaluation using rules

Values

- Recall $\mathsf{L}_{\mathsf{Arith}}$ expressions:

$$Expr \ni e ::= e_1 + e_2 \mid e_1 \times e_2 \mid n \in \mathbb{N}$$

- Some expressions, like 1,2,3, are special
- They have no remaining "computation" to do
- We call such expressions *values*.
- We can define a BNF grammar rule for values:

$$Value \ni v ::= n \in \mathbb{N}$$

Evaluation, informally

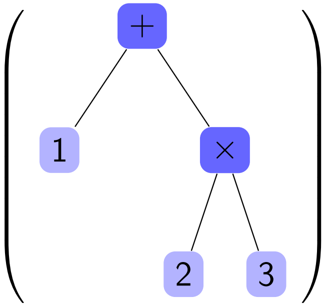
- Given an expression e, what is its value?
  - If $e = n$, a number, then it is already a value.
  - If $e = e_1 + e_2$, evaluate e_1 to v_1 and e_2 to v_2. Then add v_1 and v_2, the result is the value of e.
  - If $e = e_1 \times e_2$, evaluate e_1 to v_1 and e_2 to v_2. Then multiply v_1 and v_2, the result is the value of e.

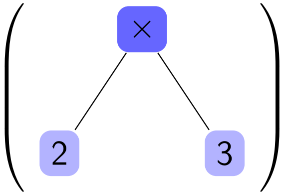
Evaluation, in Scala

- If $e = n$, a number, then it is already a value.
- If $e = e_1 + e_2$, evaluate e_1 to v_1 and e_2 to v_2. Then add v_1 and v_2, the result is the value of e.
- If $e = e_1 \times e_2$, evaluate e_1 to v_1 and e_2 to v_2. Then multiply v_1 and v_2, the result is the value of e.

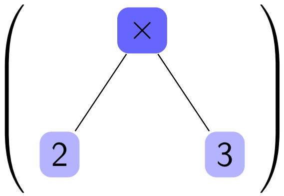
```
def eval(e: Expr): Int = e match {
  case Num(n) => n
  case Plus(e1,e2) => eval(e1) + eval(e2)
  case Times(e1,e2) => eval(e1) * eval(e2)
}
```


Example

eval 

$= eval(1) + eval$ 

Example

$eval(1) + eval$ 

$= eval(1) + (eval(2) \times eval(3))$

$$eval(1) + (eval(2) \times eval(3)) = 1 + (2 \times 3) = 1 + 6 = 7$$

Expression evaluation, more formally

- To specify and reason about evaluation, we use a *evaluation judgment*.

Definition (Evaluation judgment)

Given expression e and value v, we say v is the value of e if evaluating e results in v, and we write $e \Downarrow v$ to indicate this.

- (A *judgment* is a relation between abstract syntax trees.)
- Examples:

$$1 + 2 \Downarrow 3 \qquad 1 + 2 \times 3 \Downarrow 7 \qquad (1 + 2) \times 3 \Downarrow 9$$

Evaluation of Values

- A value is already evaluated. So, for any v, we have $v \Downarrow v$.
- We can express the fact that $v \Downarrow v$ always holds (for any v) as follows:

$$\frac{}{v \Downarrow v}$$

- This is a *rule* that says that v evaluates to v always (no preconditions)
- So, for example, we can derive:

$$\frac{}{0 \Downarrow 0} \qquad \frac{}{1 \Downarrow 1} \qquad \cdots$$

Evaluation of Addition

- How to evaluate expression $e_1 + e_2$?
- Suppose we know that $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$.
- Then the value of $e_1 + e_2$ is the number we get by adding numbers v_1 and v_2.
- We can express this as follows:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2}$$

- This is a *rule* that says that $e_1 + e_2$ evaluates to $v_1 +_{\mathbb{N}} v_2$ provided e_1 evaluates to v_1 and e_2 evaluates to v_2
- Note that we write $+_{\mathbb{N}}$ for the *mathematical function* that adds two numbers, to avoid confusion with the *abstract syntax tree* $v_1 + v_2$.

Expression evaluation: Summary

- Multiplication can be handled exactly like addition.
- We will define the meaning of L_{Arith} expressions using the following rules:

$\boxed{e \Downarrow v}$

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \times e_2 \Downarrow v_1 \times_{\mathbb{N}} v_2}$$

- This evaluation judgment is an example of *big-step semantics* (or *natural semantics*)
  - so-called because we evaluate the whole expression "in one step"

Examples

- We can use these rules to *derive* evaluation judgments for complex expressions:

$$\frac{\frac{}{1 \Downarrow 1} \quad \frac{}{2 \Downarrow 2}}{1 + 2 \Downarrow 3} \qquad \frac{\frac{}{1 \Downarrow 1} \quad \frac{\frac{}{2 \Downarrow 2} \quad \frac{}{3 \Downarrow 3}}{2 * 3 \Downarrow 6}}{1 + (2 * 3) \Downarrow 7} \qquad \frac{\frac{\frac{}{1 \Downarrow 1} \quad \frac{}{2 \Downarrow 2}}{1 + 2 \Downarrow 3} \quad \frac{}{3 \Downarrow 3}}{(1 + 2) * 3 \Downarrow 9}$$

- These figures are *derivation trees* showing how we can derive a conclusion from axioms
- The rules govern how we can construct derivation trees.
  - A leaf node must match a rule with no preconditions
  - Other nodes must match rules with preconditions. (Order matters.)
- Note that derivation trees "grow up" (root is at the bottom)

Totality and Structural induction

- Question: Given any expression e, does it evaluate to a value?
- To answer this question, we can use structural induction:

Induction on structure of expressions

Given a property P of expressions, if:

- $P(n)$ holds for every number $n \in \mathbb{N}$
- for any expressions e_1, e_2, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 + e_2)$ also holds
- for any expressions e_1, e_2, if $P(e_1)$ and $P(e_2)$ holds then $P(e_1 \times e_2)$ also holds

Then $P(e)$ holds for all expressions e.

Proof by structural induction

- Let's illustrate with an example

Theorem

If e is an expression, then there exists $v \in \mathbb{N}$ such that $e \Downarrow v$ holds.

Proof: Base case.

If $e = n$ then e is already a value. Take $v = n$, then we can derive

$$\frac{}{e \Downarrow n}$$

□

Proof by structural induction

Proof: Inductive case 1.

If $e = e_1 + e_2$ then suppose $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$ for some v_1, v_2. Then we can use the rule:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2}$$

to conclude that there exists $v = v_1 +_{\mathbb{N}} v_2$ such that $e \Downarrow v$ holds. □

Note that again it's important to distinguish $v_1 +_{\mathbb{N}} v_2$ (the number) from $v_1 + v_2$ the expression.

Proof by structural induction

Proof: Inductive case 2.

If $e = e_1 \times e_2$ then suppose $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$ for some v_1, v_2. Then we can use the rule:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \times e_2 \Downarrow v_1 \times_{\mathbb{N}} v_2}$$

to conclude that there exists $v = v_1 \times_{\mathbb{N}} v_2$ such that $e \Downarrow v$ holds. □

- This case is basically identical to case 1 (modulo $+$ vs. $\times$).
- From now on we will typically skip over such "essentially identical" cases (but it is important to really check them).

Uniqueness

We can also prove the uniqueness of the value of v by induction:

Theorem (Uniqueness of evaluation)

If $e \Downarrow v$ and $e \Downarrow v'$, then $v = v'$.

Base case.

If $e = n$ then since $n \Downarrow v$ and $n \Downarrow v'$ hold, the only way we could derive these judgments is for v, v' to both equal n. □

Uniqueness

Inductive case.

If $e = e_1 + e_2$ then the derivations must be of the form

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1' \quad e_2 \Downarrow v_2'}{e_1 + e_2 \Downarrow v_1' +_{\mathbb{N}} v_2'}$$

By induction, $e_1 \Downarrow v_1$ and $e_1 \Downarrow v_1'$ implies $v_1 = v_1'$, and similarly for e_2 so $v_2 = v_2'$. Therefore $v_1 +_{\mathbb{N}} v_2 = v_1' +_{\mathbb{N}} v_2'$. □

- The proof for $e_1 \times e_2$ is similar.

Totality, uniqueness, and correctness

- The Scala interpreter code defined earlier says how to interpret a L_{Arith} expression as a *function*
- The big-step rules, in contrast, specify the meaning of expressions as a *relation*.
- Nevertheless, *totality* and *uniqueness* guarantee that for each e there is a unique v such that $e \Downarrow v$
- In fact, $v = eval(e)$, that is:

Theorem (Interpreter Correctness)

For any L_{Arith} expression e, we have $e \Downarrow v$ if and only if $v = eval(e)$.

- Proof: induction on e.

Summary

- In this lecture, we've covered:
  - A simple interpreter
  - Evaluation via rules
  - Totality and uniqueness (via structural induction)
- all for the simple language L_{Arith}
- Next time:
  - Booleans, equality, conditionals
  - Types

Elements of Programming Languages

Lecture 3: Booleans, conditionals, and types

James Cheney

University of Edinburgh

September 30, 2016

Boolean expressions

- So far we've considered only a trivial arithmetic language $\mathsf{L}_{\mathsf{Arith}}$
- Let's extend $\mathsf{L}_{\mathsf{Arith}}$ with equality tests and Boolean true/false values:

$$e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2$$

- We write $\mathbb{B}$ for the set of Boolean values $\{\mathtt{true}, \mathtt{false}\}$
- Basic idea: $e_1 == e_2$ should evaluate to `true` if e_1 and e_2 have equal values, `false` otherwise

What use is this?

- Examples:
  - $2 + 2 == 4$ should evaluate to `true`
  - $3 \times 3 + 4 \times 4 == 5 \times 5$ should evaluate to `true`
  - $3 \times 3 == 4 \times 7$ should evaluate to `false`
  - How about $\mathtt{true} == \mathtt{true}$? Or $\mathtt{false} == \mathtt{true}$?
- So far, there's not much we can do.
- We can evaluate a numerical expression for its value, or a Boolean equality expression to true or false
- We can't write an expression whose result depends on evaluating a comparison.
  - We lack an "if then else" (conditional) operation.
- We also can't "and", "or" or negate Boolean values.

Conditionals

- Let's also add an "if then else" operation:

$$e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2 \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$$

- We define L_{If} as the extension of $\mathsf{L}_{\mathsf{Arith}}$ with booleans, equality and conditionals.
- Examples:
  - `if true then 1 else 2` should evaluate to 1
  - $\mathtt{if}\ 1 + 1 == 2\ \mathtt{then}\ 3\ \mathtt{else}\ 4$ should evaluate to 3
  - `if true then false else true` should evaluate to `false`
- Note that $\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ is the first expression that makes nontrivial "choices": whether to evaluate the first or second case.

Extending evaluation

- We consider the Boolean values `true` and `false` to be *values*:

$$v ::= n \in \mathbb{N} \mid b \in \mathbb{B}$$

- and we add the following evaluation rules:

$\boxed{e \Downarrow v}$ for L_{If}

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v}{e_1 == e_2 \Downarrow \mathtt{true}} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \neq v_2}{e_1 == e_2 \Downarrow \mathtt{false}}$$

$$\frac{e \Downarrow \mathtt{true} \quad e_1 \Downarrow v_1}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_1} \qquad \frac{e \Downarrow \mathtt{false} \quad e_2 \Downarrow v_2}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v_2}$$

Extending the interpreter

- To interpret L_{If}, we need new expression forms:

```
case class Bool(n: Boolean) extends Expr
case class Eq(e1: Expr, e2:Expr) extends Expr
case class IfThenElse(e: Expr, e1: Expr, e2: Expr)
  extends Expr
```

- and different types of values (not just `Int`s):

```
  abstract class Value
  case class NumV(n: Int) extends Value
  case class BoolV(b: Boolean) extends Value
```

- (Technically, we could encode booleans as integers, but in general we will want to separate out the kinds of values.)

Extending the interpreter

```
// helpers
def add(v1: Value, v2: Value): Value =
       (v1,v2) match {
         case (NumV(v1), NumV(v2)) => NumV (v1 + v2)
       }
def mult(v1: Value, v2: Value): Value = ...
def eval(e: Expr): Value = e match {
   // Arithmetic
   case Num(n) => NumV(n)
   case Plus(e1,e2) => add(eval(e1),eval(e2))
   case Times(e1,e2) => mult(eval(e1),eval(e2))
   ... }
```


Extending the interpreter

```
// helper
def eq(v1: Value, v2: Value): Value = (v1,v2) match {
    case (NumV(n1), NumV(n2)) => BoolV(n1 == n2)
    case (BoolV(b1), BoolV(b2)) => BoolV(b1 == b2)
}
def eval(e: Expr): Value = e match {
  ...
  case Bool(b) => BoolV(b)
  case Eq(e1,e2) => eq (eval(e1), eval(e2))
  case IfThenElse(e,e1,e2) => eval(e) match {
    case BoolV(true) => eval(e1)
    case BoolV(false) => eval(e2)
  }
}
```

Aside: Other Boolean operations

- We can add Boolean and, or and not operations as follows:

$$e ::= \cdots \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg(e)$$

- with evaluation rules:

$\boxed{e \Downarrow v}$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \wedge e_2 \Downarrow v_1 \wedge_{\mathbb{B}} v_2} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \vee e_2 \Downarrow v_1 \vee_{\mathbb{B}} v_2}$$

- where again, $\wedge_{\mathbb{B}}$ and $\vee_{\mathbb{B}}$ are the mathematical "and" and "or" operations
- These are definable in L_{If}, so we will leave them out to avoid clutter.

Aside: Shortcut operations

- Many languages (e.g. C, Java) offer *shortcut* versions of "and" and "or":

$$e ::= \cdots \mid e_1 \mathbin{\texttt{\&\&}} e_2 \mid e_1 \mathbin{\texttt{||}} e_2$$

- e_1 && e_2 stops early if e_1 is false (since e_2's value then doesn't matter).
- e_1 || e_2 stops early if e_1 is true (since e_2's value then doesn't matter).
- We can model their semantics using rules like this:

$$\frac{e_1 \Downarrow \texttt{false}}{e_1 \mathbin{\texttt{\&\&}} e_2 \Downarrow \texttt{false}} \qquad \frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow v_2}{e_1 \mathbin{\texttt{\&\&}} e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \texttt{true}}{e_1 \mathbin{\texttt{||}} e_2 \Downarrow \texttt{true}} \qquad \frac{e_1 \Downarrow \texttt{false} \quad e_2 \Downarrow v_2}{e_1 \mathbin{\texttt{||}} e_2 \Downarrow v_2}$$

What else can we do?

- We can also do strange things like this:

$$e_1 = 1 + (2 == 3)$$

- Or this:

$$e_2 = \texttt{if } 1 \texttt{ then } 2 \texttt{ else } 3$$

What should these expressions evaluate to?

- There is no v such that $e_1 \Downarrow v$ or $e_2 \Downarrow v$!
  - the *Totality* property for L_{Arith} fails, for L_{If}!
- If we try to run the interpreter: we just get an error

One answer: Conversions

- In some languages (notably C, Java), there are built-in *conversion rules*
  - For example, "if an integer is needed and a boolean is available, convert `true` to 1 and `false` to 0"
  - Likewise, "if a boolean is needed and an integer is available, convert 0 to `false` and other values to `true`"
  - LISP family languages have a similar convention: if we need a Boolean value, `nil` stands for "false" and any other value is treated as "true"
- Conversion rules are convenient but can make programs less predictable
- We will avoid them for now, but consider principled ways of providing this convenience later on.

Another answer: Types

- Should programs like:

  $1 + (2 == 3)$ `if 1 then 2 else 3`

  even be allowed?
- Idea: use a *type system* to define a subset of "well-formed" programs
- Well-formed means (at least) that at run time:
  - arguments to arithmetic operations (and equality tests) should be numeric values
  - arguments to conditional tests should be Boolean values

Typing rules, informally: arithmetic

- Consider an expression e
  - If $e = n$, then e has type "integer"
  - If $e = e_1 + e_2$, then e_1 and e_2 must have type "integer". If so, e has type "integer" also, else error.
  - If $e = e_1 \times e_2$, then e_1 and e_2 must have type "integer". If so, e has type "integer" also, else error.

Typing rules, informally: booleans, equality and conditionals

- Consider an expression e
  - If $e = \texttt{true}$ or $\texttt{false}$, then e has type "boolean"
  - If $e = e_1 == e_2$, then e_1 and e_2 must have **the same type**. If so, e has type "boolean", else error.
  - If $e = \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2$, then e_0 must have type "boolean", and e_1 and e_2 must have **the same type**. If so, then e has the same type as e_1 and e_2, else error.
- Note 1: Equality arguments have the same (unknown) type.
- Note 2: Conditional branches have the same (unknown) type. This type determines the type of the whole conditional expression.

Concise notation for typing rules

- We can define the possible types using a BNF grammar, as follows:

  $$Type \ni \tau ::= \texttt{int} \mid \texttt{bool}$$

  For now, we will consider only two possible types, "integer" ($\texttt{int}$) and "boolean" ($\texttt{bool}$).
- We can also use *rules* to describe the types of expressions:

Definition (Typing judgment $\vdash e : \tau$)

We use the notation $\vdash e : \tau$ to say that e is a well-formed term of type τ (or "e has type τ").

Typing rules, more formally: arithmetic

- If $e = n$, then e has type "integer"
- If $e = e_1 + e_2$, then e_1 and e_2 must have type "integer". If so, e has type "integer" also, else error.
- If $e = e_1 \times e_2$, then e_1 and e_2 must have type "integer". If so, e has type "integer" also, else error.

$\boxed{\vdash e : \tau}$ for $\mathrm{L}_{\mathrm{Arith}}$

$$\frac{n \in \mathbb{N}}{\vdash n : \mathtt{int}} \qquad \frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 + e_2 : \mathtt{int}}$$

$$\frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 \times e_2 : \mathtt{int}}$$

Typing rules, more formally: equality and conditionals

$\boxed{\vdash e : \tau}$ for L_{If}

$$\frac{b \in \mathbb{B}}{\vdash b : \mathtt{bool}} \qquad \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 == e_2 : \mathtt{bool}}$$

$$\frac{\vdash e : \mathtt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

- We indicate that the types of subexpressions of $==$ must be equal by using the same τ
- Similarly, we indicate that the result of a conditional has the same type as the two branches using the same τ for all three

Typing judgments: examples

$$\frac{\dfrac{\overline{\vdash 1 : \mathtt{int}} \quad \overline{\vdash 2 : \mathtt{int}}}{\vdash 1 + 2 : \mathtt{int}} \qquad \overline{\vdash 4 : \mathtt{int}}}{\vdash 1 + 2 == 4 : \mathtt{bool}}$$

$$\frac{\begin{array}{c}\vdots \\ \vdash 1 + 2 == 4 : \mathtt{bool}\end{array} \quad \overline{\vdash 42 : \mathtt{int}} \quad \overline{\vdash 17 : \mathtt{int}}}{\vdash \mathtt{if}\ 1 + 2 == 4\ \mathtt{then}\ 42\ \mathtt{else}\ 17 : \mathtt{int}}$$

$$\frac{\begin{array}{c}\vdots \\ \vdash \mathtt{if}\ 1 + 2 == 4\ \mathtt{then}\ 42\ \mathtt{else}\ 17 : \mathtt{int}\end{array} \quad \overline{\vdash 100 : \mathtt{int}}}{\vdash (\mathtt{if}\ 1 + 2 == 4\ \mathtt{then}\ 42\ \mathtt{else}\ 17) + 100 : \mathtt{int}}$$

Typing judgments: non-examples

But we also want some things **not** to typecheck:

$$\vdash 1 == \mathtt{true} : \tau$$

$$\vdash \mathtt{if}\ 42\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau$$

These judgments do not hold for any e_1, e_2, τ.

Fundamental property of typing

- The point of the typing judgment is to ensure *soundness*: if an expression is well-typed, then it evaluates "correctly"
- That is, evaluation is well-behaved on well-typed programs.

Theorem (Type soundness for L_{If})

If $\vdash e : \tau$ then $e \Downarrow v$ and $\vdash v : \tau$.

- For a language like L_{If}, soundness is fairly easy to prove by induction on expressions. We'll present soundness for more realistic languages in detail later.

Static vs. dynamic typing

- Some languages proudly advertise that they are "static" or "dynamic"
- **Static typing:**
  - not all expressions are well-formed; some sensible programs are not allowed
  - types can be used to catch errors, improve performance
- **Dynamic typing:**
  - all expressions are well-formed; any program can be run
  - type errors arise dynamically; higher overhead for tagging and checking
- These are rarely-realized extremes: most "statically" typed languages handle some errors dynamically
- In contrast, any "dynamically" typed language can be thought of as a statically typed one with just one type.

Summary

- In this lecture we covered:
  - Boolean values, equality tests and conditionals
  - Extending the interpreter to handle them
  - Typing rules
- Next time:
  - Variables and let-binding
  - Substitution, environments and type contexts

Elements of Programming Languages

Lecture 4: Variables, scope, and substitution

James Cheney

University of Edinburgh

October 4, 2016

Variables

- A variable is a symbol that can 'stand for' a value.
- Often written $x, y, z, \ldots$.
- Let's extend L_{If} with variables:

$$\begin{array}{rcl} e & ::= & n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 \times e_2 \\ & \mid & b \in \mathbb{B} \mid e_1 == e_2 \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & x \in \mathit{Var} \end{array}$$

- Here, x is shorthand for an arbitrary variable in Var, the set of expression variables
- Let's call this language $\mathsf{L}_{\mathsf{Var}}$

Aside: Operators, operators everywhere

- We have now considered several *binary operators*

$$+ \quad \times \quad \wedge \quad \vee \quad \approx$$

- as well as a unary one $(\neg)$
- It is tiresome to write their syntax, evaluation rules, and typing rules explicitly, every time we add to the language
- We will sometimes represent such operations using *schematic* syntax $e_1 \oplus e_2$ and rules:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \oplus e_2 \Downarrow v_1 \oplus_{\mathbb{A}} v_2} \qquad \frac{\vdash e_1 : \tau' \quad \vdash e_2 : \tau' \quad \oplus : \tau' \times \tau' \to \tau}{\vdash e_1 \oplus e_2 : \tau}$$

- where $\oplus : \tau' \times \tau' \to \tau$ means that operator $\oplus$ takes arguments τ', τ' and yields result of type τ
- (e.g. $+ : \texttt{int} \times \texttt{int} \to \texttt{int}$, $== : \tau \times \tau \to \texttt{bool}$)

Substitution

- We said "A variable can 'stand for' a value."
- What does this mean precisely?
- Suppose we have $x + 1$ and we want x to "stand for" 42.
- We should be able to *replace* x everywhere in $x + 1$ with 42:

$$x + 1 \rightsquigarrow 42 + 1$$

- Similarly, if x "stands for" 3 then

$$\texttt{if}\ x == y\ \texttt{then}\ x\ \texttt{else}\ y \rightsquigarrow \texttt{if}\ 3 == y\ \texttt{then}\ 3\ \texttt{else}\ y$$

Substitution

- Let's introduce a notation for this *substitution* operation:

Definition (Substitution)

Given e, x, v, the *substitution of v for x in e* is an expression written $e[v/x]$.

- For L_{Var}, define substitution as follows:

$$
\begin{array}{rcl}
v_0[v/x] & = & v_0 \\
x[v/x] & = & v \\
y[v/x] & = & y \quad (x \neq y) \\
(e_1 \oplus e_2)[v/x] & = & e_1[v/x] \oplus e_2[v/x] \\
(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2)[v/x] & = & \texttt{if } e[v/x] \texttt{ then } e_1[v/x] \\
 & & \texttt{else } e_2[v/x]
\end{array}
$$

Scope

- As we all know from programming, we can *reuse* variable names:

```
def foo(x: Int) = x + 1
def bar(x: Int) = x * x
```

- The occurrences of `x` in `foo` have nothing to do with those in `bar`
- Moreover the following code is equivalent (since `y` is not already in use in `foo` or `bar`):

```
def foo(x: Int) = x + 1
def bar(y: Int) = y * y
```


Scope

Definition (Scope)

The *scope* of a variable name is the collection of program locations in which occurrences of the variable refer to the same thing.

- I am being a little casual here: "refer to the same thing" doesn't necessarily mean that the two variable occurrences evaluate to the same value at run time.
- For example, the variables could refer to a shared *reference cell* whose value changes over time.

Scope, Binding and Bound Variables

- Certain occurrences of variables are called *binding*
- Again, consider

```
def foo(x: Int) = x + 1
def bar(y: Int) = y * y
```

- The occurrences of `x` and `y` on the left-hand side of the definitions are *binding*
- Binding occurrences define scopes: the occurrences of `x` and `y` on the right-hand side are *bound*
- Any variables not in scope of a binder are called *free*
- Key idea: Renaming all binding and bound occurrences in a scope *consistently* (avoiding name clashes) should not affect meaning

Dynamic vs. static scope

- The terms *static* and *dynamic* scope are sometimes used.
- In **static scope**, the scope and binding occurrences of all variables can be determined from the program text, **without** actually running the program.
- In **dynamic scope**, this is not necessarily the case: the scope of a variable can depend on the context in which it is evaluated **at run time**.
- We will have more to say about this later when we cover functions
  - but for now, the short version is: Static scope good, dynamic scope bad.

Simple scope: let-binding

- For now, we consider a very basic form of scope: let-binding.

$$e ::= \cdots \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

- We define L_{Let} to be L_{If} extended with variables and `let`.
- In an expression of the form $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, we say that x is *bound* in e_2
- Intuition: let-binding allows us to use a variable x as an abbreviation for some other expression:

$$\texttt{let}\ x = 1 + 2\ \texttt{in}\ 3 \times x \rightsquigarrow 3 \times (1 + 2)$$

Equivalence up to consistent renaming

- We wish to consider expressions *equivalent* if they have the same binding structure
- We can *rename* bound names to get equivalent expressions:

$$\texttt{let}\ x = y + z\ \texttt{in}\ x == w \equiv \texttt{let}\ u = y + z\ \texttt{in}\ u == w$$

- But some renamings change the binding structure:

$$\texttt{let}\ x = y + z\ \texttt{in}\ x == w \not\equiv \texttt{let}\ w = y + z\ \texttt{in}\ w == w$$

- Intuition: Renaming to u is fine, because u is not already "in use".
- But renaming to w changes the binding structure, since w was already "in use".

Freshness

- We say that a variable x is *fresh* for an expression e if there are no free occurrences of x in e.
- We can define this using rules as follows:

$x \# e$

$$\frac{}{x \# v} \qquad \frac{x \neq y}{x \# y} \qquad \frac{x \# e_1 \quad x \# e_2}{x \# e_1 \oplus e_2} \qquad \frac{x \# e \quad x \# e_1 \quad x \# e_2}{x \# \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{x \# e_1}{x \# \texttt{let}\ x = e_1\ \texttt{in}\ e_2} \qquad \frac{x \neq y \quad x \# e_1 \quad x \# e_2}{x \# \texttt{let}\ y = e_1\ \texttt{in}\ e_2}$$

- Examples:

$$x \# \texttt{true} \qquad x \# y \qquad x \# \texttt{let}\ x = 1\ \texttt{in}\ x$$

Renaming

- We will also use the following *swapping* operation to rename variables:

$$\begin{array}{rcl}
x(y\leftrightarrow z) & = & \begin{cases} y & \text{if } x = z \\ z & \text{if } x = y \\ x & \text{otherwise} \end{cases} \\
v(y\leftrightarrow z) & = & v \\
(e_1 \oplus e_2)(y\leftrightarrow z) & = & e_1(y\leftrightarrow z) \oplus e_2(y\leftrightarrow z) \\
(\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2)(y\leftrightarrow z) & = & \texttt{if } e(y\leftrightarrow z) \texttt{ then } e_1(y\leftrightarrow z) \\
 & & \texttt{else } e_2(y\leftrightarrow z) \\
(\texttt{let } x = e_1 \texttt{ in } e_2)(y\leftrightarrow z) & = & \texttt{let } x(y\leftrightarrow z) = e_1(y\leftrightarrow z) \\
 & & \texttt{in } e_2(y\leftrightarrow z)
\end{array}$$

- Example:

$$(\texttt{let } x = y \texttt{ in } x + z)(x\leftrightarrow z) = \texttt{let } z = y \texttt{ in } z + x$$

Alpha-conversion

- We can now define "consistent renaming".
- Suppose $y \# e_2$. Then we can rename a `let`-expression as follows:

$$\texttt{let } x = e_1 \texttt{ in } e_2 \rightsquigarrow_\alpha \texttt{let } y = e_1 \texttt{ in } e_2(x\leftrightarrow y)$$

- This is called *alpha-conversion*.
- Two expressions are *alpha-equivalent* if we can convert one to the other using alpha-conversions.

Examples

- Examples:

$$\begin{array}{ll}
 & \texttt{let } x = y + z \texttt{ in } x == w \\
\rightsquigarrow_\alpha & \texttt{let } u = y + z \texttt{ in } (x == w)(x\leftrightarrow u) \\
= & \texttt{let } u = y + z \texttt{ in } u(x\leftrightarrow u) == w(x\leftrightarrow u) \\
= & \texttt{let } u = y + z \texttt{ in } u == w
\end{array}$$

since $u \# (x == w)$.

- But

$$\texttt{let } x = y+z \texttt{ in } x == w \not\rightsquigarrow_\alpha \texttt{let } w = y+z \texttt{ in } w == w$$

because w already appears in $x == w$.

Types and variables

- Once we add variables to our language, how does that affect typing?
- Consider

$$\texttt{let } x = e_1 \texttt{ in } e_2$$

When is this well-formed? What type does it have?

- Consider a variable on its own: what type does it have?
- **Different occurrences of the same variable in different scopes could have different types.**
- We need a way to *keep track of* the types of variables

Types for variables and let, informally

- Suppose we have a way of keeping track of the types of variables (say, some kind of map or table)
- When we see a variable x, look up its type in the map.
- When we see a $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, find out the type of e_1. Suppose that type is τ_1. Add the information that x has type τ_1 to the map, and check e_2 using the augmented map.
- Note: The local information about x's type should not persist beyond typechecking its scope e_2.

Types for variables and let, informally

- For example:

$$\texttt{let}\ x = 1\ \texttt{in}\ x + 1$$

  is well-formed: we know that x must be an `int` since it is set equal to 1, and then $x + 1$ is well-formed because x is an `int` and 1 is an `int`.
- On the other hand,

$$\texttt{let}\ x = 1\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ 42\ \texttt{else}\ 17$$

  is not well-formed: we again know that x must be an `int` while checking $\texttt{if}\ x\ \texttt{then}\ 42\ \texttt{else}\ 17$, but then when we check that the conditional's test x is a `bool`, we find that it is actually an `int`.

Type Environments

- We write Γ to denote a *type environment*, or a finite map from variable names to types, often written as follows:

$$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n$$

- In Scala, we can use the built-in type `ListMap[Variable,Type]` for this.
  - *hey, maybe that's why the Lab has all that stuff about `ListMaps`!*
- Moreover, we write $\Gamma(x)$ for the type of x according to Γ and $\Gamma, x : \tau$ to indicate extending Γ with the mapping x to τ.

Types for variables and let, formally

- We now generalize the ideal of well-formedness:

Definition (Well-formedness in a context)

We write $\Gamma \vdash e : \tau$ to indicate that e is well-formed at type τ (or just "has type τ") in context Γ.

- The rules for variables and let-binding are as follows:

$\boxed{\Gamma \vdash e : \tau}$ for $\mathrm{L}_{\mathrm{Let}}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Types for variables and let, formally

- We also need to generalize the L_{If} rules to allow contexts:

$\boxed{\Gamma \vdash e : \tau}$ for L_{If}

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

$$\frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

- This is straightforward: we just add Γ everywhere.
- The previous rules are special cases where Γ is empty.

Examples, revisited

We can now typecheck as follows:

$$\frac{\frac{}{\vdash 1 : \mathtt{int}} \quad \frac{\frac{}{x : \mathtt{int} \vdash x : \mathtt{int}} \quad \frac{}{x : \mathtt{int} \vdash 1 : \mathtt{int}}}{x : \mathtt{int} \vdash x + 1 : \mathtt{int}}}{\vdash \mathtt{let}\ x = 1\ \mathtt{in}\ x + 1 : \mathtt{int}}$$

On the other hand:

$$\frac{\frac{}{\vdash 1 : \mathtt{int}} \quad \frac{x : \mathtt{int} \vdash x : \mathtt{bool} \quad \cdots}{x : \mathtt{int} \vdash \mathtt{if}\ x\ \mathtt{then}\ 42\ \mathtt{else}\ 17 : ??}}{\vdash \mathtt{let}\ x = 1\ \mathtt{in}\ \mathtt{if}\ x\ \mathtt{then}\ 42\ \mathtt{else}\ 17 : ??}$$

is not derivable because the judgment $x : \mathtt{int} \vdash x : \mathtt{bool}$ isn't.

Evaluation for `let` and variables

- One approach: whenever we see $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$,
  1. evaluate e_1 to v_1
  2. replace x with v_1 in e_2 and evaluate that

$\boxed{e \Downarrow v}$ for L_{Let}

$$\frac{e_1 \Downarrow v_1 \quad e_2[v_1/x] \Downarrow v_2}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v_2}$$

- Note: We always substitute values for variables, and do not need a rule for "evaluating" a variable
- This evaluation strategy is called *eager*, *strict*, or (for historical reasons) *call-by-value*
- This is a design choice. We will revisit this choice (and consider alternatives) later.

Substitution-based interpreter

```
type Variable = String
...
case class Var(x: Variable) extends Expr
case class Let(x: Variable, e1: Expr, e2: Expr)
  extends Expr
...
def eval(e: Expr): Value = e match {
  ...
  case Let(x,e1,e2) => {
    val v = eval(e1);
    val e2vx = subst(e2,v,x);
    eval(e2vx)
  }
```

- Note: No case for `Var(x)`.

Alternative semantics: environments

- Another common way to handle variables is to use an *environment*
- An environment σ is a partial function from variables to values (e.g. a Scala `ListMap[Variable,Value]`).
- We add σ as an argument to the evaluation judgment:

$\sigma, e \Downarrow v$

$$\frac{}{\sigma, v \Downarrow v} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 \times e_2 \Downarrow v_1 \times_{\mathbb{N}} v_2}$$

$$\cdots \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma[x = v], e_2 \Downarrow v_2}{\sigma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v_2} \qquad \frac{}{\sigma, x \Downarrow \sigma(x)}$$

- Assignment 2 will ask you to implement such an interpreter.

Summary

- Today we've covered:
  - Variables that can be replaced with values
  - Scope and binding, alpha-equivalence
  - Let-binding and how it affects typing and semantics

  Next time:
  - Functions and function types
  - Recursion

Elements of Programming Languages

Lecture 5: Functions and recursion

James Cheney

University of Edinburgh

October 7, 2016

Overview

- So far, we've covered
  - arithmetic
  - booleans, conditionals (`if then else`)
  - variables and simple binding (`let`)
- L_{Let} allows us to compute values of expressions
- and use variables to store intermediate values
- but not to define *computations* on unknown values.
- That is, there is no feature analogous to Haskell's functions, Scala's `def`, or methods in Java.
- Today, we consider *functions* and *recursion*

Named functions

- A simple way to add support for functions is as follows:

$$e ::= \cdots \mid f(e) \mid \mathtt{let\ fun}\ f(x:\tau) = e_1\ \mathtt{in}\ e_2$$

- Meaning: Define a function called f that takes an argument x and whose result is the expression e_1.
- Make f available for use in e_2.
- (That is, the scope of x is e_1, and the scope of f is e_2.)
- This is pretty limited:
  - for now, we consider one-argument functions only.
  - no recursion
  - functions are not first-class "values" (e.g. can't pass a function as an argument to another)

Examples

- We can define a squaring function:

$$\mathtt{let\ fun}\ \mathit{square}(x:\mathtt{int}) = x \times x\ \mathtt{in}\ \cdots$$

- or (assuming inequality tests) absolute value:

$$\mathtt{let\ fun}\ \mathit{abs}(x:\mathtt{int}) = \mathtt{if}\ x < 0\ \mathtt{then}\ -x\ \mathtt{else}\ x\ \mathtt{in}\ \cdots$$

Types for named functions

- We introduce a *type constructor* $\tau_1 \to \tau_2$, meaning "the type of functions taking arguments in τ_1 and returning τ_2"
- We can typecheck named functions as follows:

$$\frac{\Gamma, x{:}\tau_1 \vdash e_1 : \tau_2 \quad \Gamma, f{:}\tau_1 \to \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let fun } f(x : \tau_1) = e_1 \texttt{ in } e_2 : \tau}$$

$$\frac{\Gamma(f) = \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

- For convenience, we just use a single environment Γ for both variables and function names.

Example

Typechecking of $abs(-42)$

$$\frac{\dfrac{\dfrac{\Gamma(x) = \texttt{int}}{\Gamma \vdash x : \texttt{int}} \quad \dfrac{}{\Gamma \vdash 0 : \texttt{int}}}{\Gamma \vdash x < 0 : \texttt{bool}} \quad \dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}}}{\Gamma \vdash -x : \texttt{int}} \quad \dfrac{\Gamma(x) = \texttt{int}}{\Gamma \vdash x : \texttt{int}}}{\Gamma \vdash \texttt{if } x < 0 \texttt{ then } -x \texttt{ else } x : \texttt{int}}$$

$$\frac{\dfrac{\vdots}{\Gamma \vdash e_{abs} : \texttt{int}} \quad \dfrac{\dfrac{}{abs{:}\texttt{int} \to \texttt{int} \vdash -42 : \texttt{int}}}{abs{:}\texttt{int} \to \texttt{int} \vdash abs(-42) : \texttt{int}}}{\vdash \texttt{let fun } abs(x : \texttt{int}) = e_{abs} \texttt{ in } abs(-42) : \texttt{int}}$$

where $e_{abs} = \texttt{if } x < 0 \texttt{ then } -x \texttt{ else } x$ and $\Gamma = x{:}\texttt{int}$.

Semantics of named functions

- We can define rules for evaluating named functions as follows.
- First, let δ be an environment mapping function names f to their "definitions", which we'll write as $\langle x \Rightarrow e \rangle$.
- When we encounter a function definition, add it to δ.

$$\frac{\delta[f \mapsto \langle x \Rightarrow e_1 \rangle], e_2 \Downarrow v}{\delta, \texttt{let fun } f(x : \tau) = e_1 \texttt{ in } e_2 \Downarrow v}$$

- When we encounter an application, look up the definition and evaluate the body with the argument value substituted for the argument:

$$\frac{\delta, e_0 \Downarrow v_0 \quad \delta(f) = \langle x \Rightarrow e \rangle \quad \delta, e[v_0/x] \Downarrow v}{\delta, f(e_0) \Downarrow v}$$

Examples

Evaluation of $abs(-42)$

$$\frac{\delta, -42 < 0 \Downarrow \texttt{true} \quad \delta, -(-42) \Downarrow 42}{\delta, \texttt{if } -42 < 0 \texttt{ then } -(-42) \texttt{ else } -42 \Downarrow 42}$$

$$\frac{\dfrac{\delta, -42 \Downarrow -42 \quad \delta(abs) = \langle x \Rightarrow e_{abs} \rangle \quad \dfrac{\vdots}{\delta, e_{abs}[-42/x] \Downarrow 42}}{\delta, abs(-42) \Downarrow 42}}{\texttt{let fun } abs(x : \texttt{int}) = e_{abs} \texttt{ in } abs(-42) \Downarrow 42}$$

where $e_{abs} = \texttt{if } x < 0 \texttt{ then } -x \texttt{ else } x$ and $\delta = [abs \mapsto \langle x \Rightarrow e_{abs} \rangle]$

Static vs. dynamic scope

- Function bodies can contain free variables. Consider:

$$\begin{array}{l}\texttt{let } x = 1 \texttt{ in}\\ \texttt{let fun } f(y : \texttt{int}) = x + y \texttt{ in}\\ \texttt{let } x = 10 \texttt{ in } f(3)\end{array}$$

- Here, x is bound to 1 at the time f is defined, but re-bound to 10 when by the time f is called.
- There are two reasonable-seeming result values, depending on which x is *in scope*:
  - **Static scope** uses the binding $x = 1$ present when f is **defined**, so we get $1 + 3 = 4$.
  - **Dynamic scope** uses the binding $x = 10$ present when f is **used**, so we get $10 + 3 = 13$.

Dynamic scope breaks type soundness

- Even worse, what if we do this:

$$\begin{array}{l}\texttt{let } x = 1 \texttt{ in}\\ \texttt{let fun } f(y : \texttt{int}) = x + y \texttt{ in}\\ \texttt{let } x = \texttt{true in } f(3)\end{array}$$

- When we typecheck f, x is an integer, but it is re-bound to a boolean by the time f is called.
- The program as a whole typechecks, but we get a run-time error: *dynamic scope makes the type system unsound!*
- Early versions of LISP used dynamic scope, and it is arguably useful in an untyped language.
- Dynamic scope is now generally acknowledged as a mistake — but one that naive language designers still make.

Anonymous, first-class functions

- In many languages (including Java as of version 8), we can also write an expression for a function without a name:

$$\lambda x : \tau.\ e$$

- Here, λ (Greek letter lambda) introduces an anonymous function expression in which x is bound in e.
  - (The λ-notation dates to Church's higher-order logic (1940); there are several competing stories about why he chose λ.)
- In Scala one writes: `(x: Type) => e`
- In Java 8: `x -> e` (no type needed)
- In Haskell: `\x -> e` or `\x::Type -> e`
- The *lambda-calculus* is a model of anonymous functions

Types for the λ-calculus

- We define L_{Lam} to be L_{Let} extended with typed λ-abstraction and application as follows:

$$\begin{array}{rcl} e & ::= & \cdots \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \\ \tau & ::= & \cdots \mid \tau_1 \to \tau_2 \end{array}$$

- $\tau_1 \to \tau_2$ is (again) the type of *functions from* τ_1 *to* τ_2.
- We can extend the typing rules as follows:

$\boxed{\Gamma \vdash e : \tau}$ for L_{Lam}

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Evaluation for the λ-calculus

- Values are extended to include λ-abstractions $\lambda x.\ e$:

$$v ::= \cdots \mid \lambda x.\ e$$

(Note: We elide the type annotations when not needed.)
- and the evaluation rules are extended as follows:

$\boxed{e \Downarrow v}$ for L_{Lam}

$$\frac{}{\lambda x.\ e \Downarrow \lambda x.\ e} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

- Note: Combined with `let`, this subsumes named functions! We can just define `let fun` as "syntactic sugar"

$$\texttt{let fun } f(x{:}\tau) = e_1 \texttt{ in } e_2 \iff \texttt{let } f = \lambda x{:}\tau.\ e_1 \texttt{ in } e_2$$

Examples

- In L_{Lam}, we can define a higher-order function that calls its argument twice:

$$\texttt{let fun } twice(f : \tau \to \tau) = \lambda x{:}\tau.\ f(f(x)) \texttt{ in } \cdots$$

- and we can define the composition of two functions:

$$\texttt{let } compose = \lambda f{:}\tau_2 \to \tau_3.\ \lambda g{:}\tau_1 \to \tau_2.\ \lambda x{:}\tau_1.\ f(g(x)) \texttt{ in } \cdots$$

- Notice we are using repeated λ-abstractions to handle multiple arguments

Recursive functions

- However, L_{Lam} still cannot express general recursion, e.g. the factorial function:

$$\begin{array}{l}\texttt{let fun } fact(n{:}\texttt{int}) = \\ \quad \texttt{if } n == 0 \texttt{ then } 1 \texttt{ else } n \times fact(n-1) \texttt{ in } \cdots\end{array}$$

is not allowed because *fact* is not in scope inside the function body.
- We can't write it directly as a λ-expression $\lambda x{:}\tau.\ e$ either because we don't have a "name" for the function we're trying to define inside e.

Named recursive functions

- In many languages, named function definitions are recursive by default. (C, Python, Java, Haskell, Scala)
- Others explicitly distinguish between nonrecursive and recursive (named) function definitions. (Scheme, OCaml, F#)

```
let f(x) = e      // nonrecursive:
                  // only x is in scope in e
let rec f(x) = e  // recursive:
                  // both f and x in scope in e
```

- Note: In the *untyped* λ-calculus, `let rec` is *definable* using a special λ-term called the *Y combinator*

Anonymous recursive functions

- Inspired by L_{Lam}, we introduce a notation for anonymous *recursive* functions:

$$e ::= \cdots \mid \texttt{rec}\ f(x : \tau_1) : \tau_2.\ e$$

- Idea: f is a local name for the function being defined, and is in scope in e, along with the argument x.
- We define L_{Rec} to be L_{Lam} extended with `rec`.
- We can then define `let rec` as syntactic sugar:

$$\begin{array}{l}\texttt{let rec}\ f(x{:}\tau_1) : \tau_2 = e_1\ \texttt{in}\ e_2 \\ \quad \iff \texttt{let}\ f = \texttt{rec}\ f(x{:}\tau_1) : \tau_2.\ e_1\ \texttt{in}\ e_2\end{array}$$

- Note: The outer f is in scope in e_2, while the inner one is in scope in e_1. The two f bindings are unrelated.

Anonymous recursive functions: typing

- The types of L_{Rec} are the same. We just add one rule:

$\boxed{\Gamma \vdash e : \tau}$ for L_{Rec}

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{rec}\ f(x{:}\tau_1) : \tau_2.\ e : \tau_1 \to \tau_2}$$

- This says: to typecheck a recursive function,
  - bind f to the type $\tau_1 \to \tau_2$ (so that we can call it as a function in e),
  - bind x to the type τ_1 (so that we can use it as an argument in e),
  - typecheck e.
- Since we use the same function type, the existing function application rule is unchanged.

Anonymous recursive functions: semantics

- Like a λ-term, a recursive function is a value:

$$v ::= \cdots \mid \texttt{rec}\ f(x).\ e$$

- We can evaluate recursive functions as follows:

$\boxed{e \Downarrow v}$ for L_{Rec}

$$\frac{}{\texttt{rec}\ f(x).\ e \Downarrow \texttt{rec}\ f(x).\ e}$$

$$\frac{e_1 \Downarrow \texttt{rec}\ f(x).\ e \quad e_2 \Downarrow v_2 \quad e[\texttt{rec}\ f(x).\ e/f, v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

- To apply a recursive function, we substitute the argument for x and the whole `rec` expression for f.

Examples

- We can now write, typecheck and run *fact*
  - (you will implement an evaluator for L_{Rec} in Assignment 2 that can do this)
- In fact, L_{Rec} is *Turing-complete* (though it is still so limited that it is not very useful as a general-purpose language)
- (*Turing complete* means: able to simulate any *Turing machine*, that is, any computable function / any other programming language. ITCS covers Turing completeness and computability in depth.)

Mutual recursion

- What if we want to define mutually recursive functions?
- A simple example:

```
def even(n: Int) = if n == 0 then true else odd(n-1)
def odd(n: Int) = if n == 0 then false else even(n-1)
```

  Perhaps surprisingly, we can't easily do this!
- One solution: generalize `let rec`:

  $$\texttt{let rec } f_1(x_1{:}\tau_1) : \tau_1' = e_1 \texttt{ and } \cdots \texttt{ and } f_n(x_n{:}\tau_n) : \tau_n' = e_n \\ \texttt{in } e$$

  where $f_1, \ldots, f_n$ are all in scope in bodies $e_1, \ldots, e_n$.
- This gets messy fast; we'll revisit this issue later.

Named functions Anonymous functions Recursion

Summary

- Today we have covered:
  - Named functions
  - Static vs. dynamic scope
  - Anonymous functions
  - Recursive functions
- along with our first "composite" type, the function type $\tau_1 \to \tau_2$.
- Next time
  - Data structures: Pairs (combination) and variants (choice)

Elements of Programming Languages

Lecture 6: Data structures

James Cheney

University of Edinburgh

October 11, 2016

The story so far

- We've now covered the main ingredients of any programming language:
  - Abstract syntax
  - Semantics/interpretation
  - Types
  - Variables and binding
  - Functions and recursion
- but only in the context of a very weak language: there are no "data structures" (records, lists, variants), pointers, side-effects etc.
- Let alone even more advanced features such as classes, interfaces, or generics
- Over the next few lectures we will show how to add them, consolidating understanding of the foundations along the way.

Pairs

- The simplest way to combine data structures: pairing

$$(1, 2) \qquad (\texttt{true}, \texttt{false}) \qquad (1, (\texttt{true}, \lambda x{:}\texttt{int}.x + 2))$$

- If we have a pair, we can *extract* one of the components:

$$\texttt{fst}\ (1, 2) \rightsquigarrow 1 \qquad \texttt{snd}\ (\texttt{true}, \texttt{false}) \rightsquigarrow \texttt{false}$$

$$\texttt{snd}\ (1, (\texttt{true}, \lambda x{:}\texttt{int}.x + 2)) \rightsquigarrow (\texttt{true}, \lambda x{:}\texttt{int}.x + 2)$$

- Finally, we can often *pattern match* against a pair, to extract both components at once:

$$\texttt{let pair}\ (x, y) = (1, 2)\ \texttt{in}\ (y, x) \rightsquigarrow (2, 1)$$

Pairs in various languages

Haskell	Scala	Java	Python
`(1,2)`	`(1,2)`	`new Pair(1,2)`	`(1,2)`
`fst e`	`e._1`	`e.getFirst()`	`e[0]`
`snd e`	`e._2`	`e.getSecond()`	`e[1]`
`let (x,y) =`	`val (x,y) =`	N/A	N/A

- Functional languages typically have explicit syntax (and types) for pairs
- Java and C-like languages have "record", "struct" or "class" structures that accommodate multiple, named fields.
  - A pair type can be defined but is not built-in and there is no support for pattern-matching

Syntax and Semantics of Pairs

- Syntax of pair expressions and values:

$$\begin{array}{rcl} e & ::= & \cdots \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\ & \mid & \texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 \\ v & ::= & \cdots \mid (v_1, v_2) \end{array}$$

$\boxed{e \Downarrow v}$ for pairs

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{snd}\ e \Downarrow v_2}$$

$$\frac{e_1 \Downarrow (v_1, v_2) \quad e_2[v_1/x, v_2/y] \Downarrow v}{\texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

Types for Pairs

- Types for pair expressions:

$$\tau \quad ::= \quad \cdots \mid \tau_1 \times \tau_2$$

$\boxed{\Gamma \vdash e : \tau}$ for pairs

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2 : \tau}$$

`let` vs. `fst` and `snd`

- The `fst` and `snd` operations are definable in terms of `let pair`:

$$\begin{array}{lcl} \texttt{fst}\ e & \iff & \texttt{let pair}\ (x, y) = e\ \texttt{in}\ x \\ \texttt{snd}\ e & \iff & \texttt{let pair}\ (x, y) = e\ \texttt{in}\ y \end{array}$$

- Actually, the `let pair` construct is definable in terms of `let`, `fst`, `snd` too:

$$\begin{array}{l} \texttt{let pair}\ (x, y) = e\ \texttt{in}\ e_2 \\ \quad \iff \texttt{let}\ p = e\ \texttt{in}\ e_2[\texttt{fst}\ p/x, \texttt{snd}\ p/y] \end{array}$$

- We typically just use the (simpler) `fst` and `snd` constructs and treat `let pair` as syntactic sugar.

More generally: tuples and records

- Nothing stops us from adding triples, quadruples, ..., n-tuples.

$$(1, 2, 3) \qquad (\texttt{true}, 2, 3, \lambda x.(x, x))$$

- As mentioned earlier, many languages prefer *named* record syntax:

$$(a : 1, b : 2, c : 3) \qquad (b : \texttt{true}, n_1 : 2, n_2 : 3, f : \lambda x.(x, x))$$

- (cf. class fields in Java, structs in C, etc.)
- These are undeniably useful, but are definable using pairs.
- We'll revisit named record-style constructs when we consider classes and modules.

Special case: the "unit" type

- Nothing stops us from adding a type of *0-tuples*: a data structure with no data. This is often called the *unit type*, or `unit`.

$$
\begin{array}{lcl}
e & ::= & \cdots \mid () \\
v & ::= & \cdots \mid () \\
\tau & ::= & \cdots \mid \texttt{unit}
\end{array}
$$

$$
\frac{}{() \Downarrow ()} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}
$$

- this may seem a little pointless: why bother to define a type with no (interesting) data and no operations?
- This is analogous to `void` in C/Java; in Haskell and Scala it is called `()`.

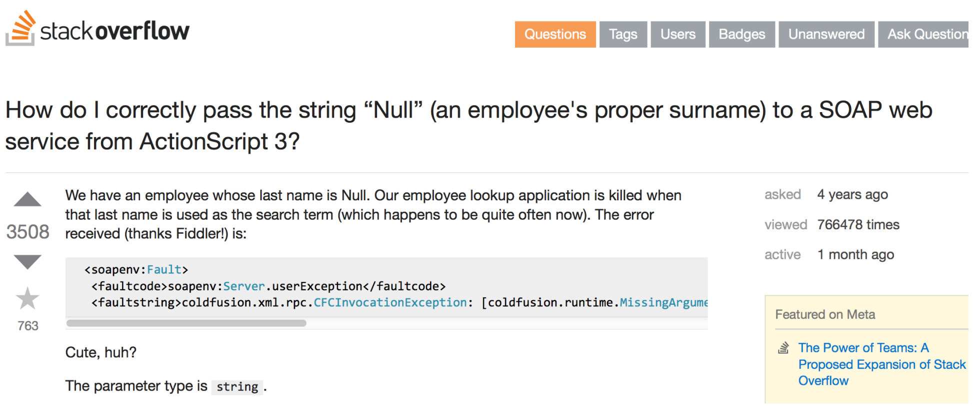
Motivation for variant types

- Pairs allow us to combine two data structures (a τ_1 and a τ_2).
- What if we want a data structure that allows us to *choose* between different options?
- We've already seen one example: booleans.
  - A boolean can be one of two values.
  - Given a boolean, we can look at its value and choose among two options, using `if then else`.
- Can we generalize this idea?

Another example: null values

- Sometimes we want to produce *either* a regular value *or* a special "null" value.
- Some languages, including SQL and Java, allow many types to have null values by default.
  - This leads to the need for defensive programming to avoid the dreaded `NullPointerException` in Java, or strange query behavior in SQL
  - Sir Tony Hoare (inventor of Quicksort) introduced null references in Algol in 1965 "simply because it was so easy to implement"!
  - he now calls them "the billion dollar mistake": `http://www.infoq.com/presentations/`↩ `Null-References-The-Billion`↩ `-Dollar-Mistake-Tony-Hoare`

Another problem with `Null`



What would be better?

- Consider an *option type*:

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \texttt{none} \mid \texttt{some}(e) \\
\tau & ::= & \cdots \mid \texttt{option}[\tau]
\end{array}
$$

$$
\frac{}{\Gamma \vdash \texttt{none} : \texttt{option}[\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{some}(e) : \texttt{option}[\tau]}
$$

- Then we can use `none` to indicate absence of a value, and $\texttt{some}(e)$ to give the present value.
- Morover, the *type* of an expression tells us whether null values are possible.

Error codes

- The option type is useful but still a little limited: we either get a τ value, or nothing
- If `none` means failure, we might want to get some more information about why the failure occurred.
- We would like to be able to return an *error code*
  - In older languages, notably C, special values are often used for errors
  - Example: `read` reads from a file, and either returns number of bytes read, or -1 representing an error
  - The actual error code is passed via a global variable
  - It's easy to forget to check this result, and the function's return value can't be used to return data.
  - Other languages use *exceptions*, which we'll cover much later

The OK-or-error type

- Suppose we want to return *either* a normal value τ_{ok} *or* an error value τ_{err}.
- Let's write $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$ for this type.

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \texttt{ok}(e) \mid \texttt{err}(e) \\
\tau & ::= & \cdots \mid \texttt{okOrErr}[\tau_1, \tau_2]
\end{array}
$$

- Basic idea:
  - if e has type τ_{ok}, then $\texttt{ok}(e)$ has type $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$
  - if e has type τ_{err}, then $\texttt{err}(e)$ has type $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$

How do we use $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$?

- When we talked about $\texttt{option}[\tau]$, we didn't really say how to *use* the results.
- If we have a $\texttt{okOrErr}[\tau_{ok}, \tau_{err}]$ value v, then we want to be able to *branch* on its value:
  - If v is $\texttt{ok}(v_{ok})$, then we probably want to get at v_{ok} and use it to proceed with the computation
  - If v is $\texttt{err}(v_{err})$, then we probably want to get at v_{err} to report the error and stop the computation.
- In other words, we want to perform *case analysis* on the value, and extract the wrapped value for further processing

Case analysis

- We consider a case analysis construct as follows:

$$\texttt{case}\ e\ \texttt{of}\ \{\texttt{ok}(x) \Rightarrow e_{ok}\ ;\ \texttt{err}(y) \Rightarrow e_{err}\}$$

- This is a generalized conditional: "If e evaluates to $\texttt{ok}(v_{ok})$, then evaluate e_{ok} with v_{ok} replacing x, else it evaluates to $\texttt{err}(v_{err})$ so evaluate e_{err} with v_{err} replacing y."
- Here, x is bound in e_{ok} and y is bound in e_{err}
- This construct should be familiar by now from Scala:

```
e match { case Ok(x) => e1
          case Err(x) => e2
} // note slightly different syntax
```


Variant types, more generally

- Notice that the ok and err cases are completely symmetric
- Generalizing this type might also be useful for other situations than error handling...
- Therefore, let's rename and generalize the notation:

$$\begin{array}{rcl} e & ::= & \cdots \mid \texttt{left}(e) \mid \texttt{right}(e) \\ & \mid & \texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \\ v & ::= & \cdots \mid \texttt{left}(v) \mid \texttt{right}(v) \\ \tau & ::= & \cdots \mid \tau_1 + \tau_2 \end{array}$$

- We will call type $\tau_1 + \tau_2$ a *variant type* (sometimes also called *sum* or *disjoint union*)

Types for variants

- We extend the typing rules as follows:

$\boxed{\Gamma \vdash \tau}$ for variant types

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{left}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{right}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} : \tau}$$

- Idea: left and right "wrap" τ_1 or τ_2 as $\tau_1 + \tau_2$
- Idea: Case is like conditional, only we can use the wrapped value extracted from $\texttt{left}(v)$ or $\texttt{right}(v)$.

Semantics of variants

- We extend the evaluation rules as follows:

$\boxed{e \Downarrow v}$ for variant types

$$\frac{e \Downarrow v}{\texttt{left}(e) \Downarrow \texttt{left}(v)} \qquad \frac{e \Downarrow v}{\texttt{right}(e) \Downarrow \texttt{right}(v)}$$

$$\frac{e \Downarrow \texttt{left}(v_1) \quad e_1[v_1/x] \Downarrow v}{\texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \Downarrow v}$$

$$\frac{e \Downarrow \texttt{right}(v_2) \quad e_2[v_2/y] \Downarrow v}{\texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\} \Downarrow v}$$

- Creating a $\tau_1 + \tau_2$ value is straightforward.
- Case analysis branches on the $\tau_1 + \tau_2$ value

Defining Booleans and option types

- The Boolean type `bool` can be defined as $\texttt{unit} + \texttt{unit}$

$$\texttt{true} \iff \texttt{left}() \qquad \texttt{false} \iff \texttt{right}()$$

- Conditional is then defined as case analysis, ignoring the variables

$$\begin{array}{l}\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ \quad \iff \texttt{case}\ e\ \texttt{of}\ \{\texttt{left}(x) \Rightarrow e_1\ ;\ \texttt{right}(y) \Rightarrow e_2\}\end{array}$$

- Likewise, the option type is definable as $\tau + \texttt{unit}$:

$$\texttt{some}(e) \iff \texttt{left}(e) \qquad \texttt{none} \iff \texttt{right}()$$

Datatypes: named variants and case classes

- Programming directly with binary variants is awkward
- As for pairs, the $\tau_1 + \tau_2$ type can be generalized to *n*-ary choices or *named variants*
- As we saw in Lecture 1 with abstract syntax trees, variants can be represented in different ways
  - Haskell supports "datatypes" which give constructor names to the cases
  - In Java, can use classes and inheritance to simulate this, verbosely (Python similar)
  - Scala does not directly support named variant types, but provides "case classes" and pattern matching
  - We'll revisit case classes and variants later in discussion of object-oriented programming.

The empty type

- We can also consider the 0-ary variant type

$$\tau \ ::= \ \cdots \mid \texttt{empty}$$

  with *no* associated expressions or values
- Scala provides `Nothing` as a built-in type; most languages do not
  - [Perhaps confusingly, this is not the same thing at all as the `void` or `unit` type!]
- We will talk about `Nothing` again when we cover *subtyping*
  - (Insert *Seinfeld* joke here, if anyone is old enough to remember that.)

Summary

- Today we've covered two primitive types for structured data:
  - Pairs, which combine two or more data structures
  - Variants, which represent alternative choices among data structures
  - Special cases (unit, empty) and generalizations (records, datatypes)
- This is a pattern we'll see over and over:
  - Define a type and expressions for creating and using its elements
  - Define typing rules and evaluation rules
- Next time:
  - Named records and variants
  - Subtyping

Elements of Programming Languages

Lecture 7: Records, variants, and subtyping

James Cheney

University of Edinburgh

October 18, 2016

Overview

- Last time:
  - Simple data structures: pairing (product types), choice (sum types)
- Today:
  - Records (generalizing products), variants (generalizing sums) and pattern matching
  - Subtyping

Records

- *Records* generalize pairs to n-tuples with *named* fields.

$$\begin{array}{rcl} e & ::= & \cdots \mid \langle l_1 = e_1, \ldots, l_n = e_n \rangle \mid e.l \\ v & ::= & \cdots \mid \langle l_1 = v_1, \ldots, l_n = v_n \rangle \\ \tau & ::= & \cdots \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \end{array}$$

- Examples:

$$\langle fst{=}1, snd{=}\texttt{"forty-two"} \rangle.snd \mapsto \texttt{"forty-two"}$$
$$\langle x{=}3.0, y{=}4.0, length{=}5.0 \rangle$$

- Record fields can be (first-class) functions too:

$$\langle x{=}3.0, y{=}4.0, length{=}\lambda(x,y).\ sqrt(x * x + y * y) \rangle$$

Named variants

- As mentioned earlier, *named variants* generalize binary variants just as records generalize pairs

$$\begin{array}{rcl} e & ::= & \cdots \mid C_i(e) \mid \texttt{case}\ e\ \texttt{of}\ \{C_1(x) \Rightarrow e_1; \ldots\} \\ v & ::= & \cdots \mid C_i(v) \\ \tau & ::= & \cdots \mid [C_1 : \tau_1, \ldots, C_n : \tau_n] \end{array}$$

- Basic idea: allow a choice of n cases, each with a name
- To construct a named variant, use the constructor name on a value of the appropriate type, e.g. $C_i(e_i)$ where $e_i : \tau_i$
- The case construct generalizes to named variants also

Named variants in Scala: case classes

- We have already seen (and used) Scala's *case class* mechanism

```
abstract class IntList
case class Nil() extends IntList
case class Cons(head: Int, tail: IntList)
  extends IntList
```

- Note: `IntList`, `Nil`, `Cons` are newly defined types, different from any others.
- Case classes support *pattern matching*

```
def foo(x: IntList) = x match {
  case Nil() => ...
  case Cons(head,tail) => ...
}
```


Aside: Records and Variants in Haskell

- In Haskell, `data` defines a recursive, named variant type

  `data IntList = Nil Int | Cons Int IntList`
- and cases can define named fields:

  `data Point = Point {x :: Double, y :: Double}`
- In both cases the newly defined type is different from any other type seen so far, and the named constructor(s) can be used in pattern matching
- This approach dates to the ML programming language (Milner et al.) and earlier designs such as HOPE (Burstall et al.).
  - (Both developed in Edinburgh)

Pattern matching

- Datatypes and case classes support *pattern matching*
  - We have seen a simple form of pattern matching for sum types.
  - This generalizes to named variants
  - But still is very limited: we only consider one "level" at a time
- Patterns typically also include constants and pairs/records

```
x match { case (1, (true, "abcd")) => ...}
```

- Patterns in Scala, Haskell, ML can also be *nested*: that is, they can match more than one constructor

```
x match { case Cons(1,Cons(y,Nil())) => ...}
```


More pattern matching

- Variables cannot be repeated, instead, explicit equality tests need to be used.
- The special pattern _ matches anything
- Patterns can overlap, and usually they are tried in order

```
result match {
  case OK => println("All␣is␣well")
  case _ => println("Release␣the␣hounds!")
}
// not the same as
result match {
  case _ => println("Release␣the␣hounds!")
  case OK => println("All␣is␣well")
}
```

Expanding nested pattern matching

- Nested pattern matching can be expanded out:

```
l match {
  case Cons(x,Cons(y,Nil())) => ...
}
```

expands to

```
l match {
  case Cons(x,t1) => t1 match {
      case Cons(y,t2) => t2 match {
        case Nil() => ...
} } }
```


Type abbreviations

- Obviously, it quickly becomes painful to write "$\langle x : \texttt{int}, y : \texttt{str}\rangle$" over and over.
- **Type abbreviations** introduce a name for a type.

$$\texttt{type } T = \tau$$

An abbreviation name T treated the same as its expansion τ

  - (much like `let`-bound variables)
- Examples:

$$\begin{array}{l} \texttt{type } \mathit{Point} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}\rangle \\ \texttt{type } \mathit{Point3d} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, z{:}\texttt{dbl}\rangle \\ \texttt{type } \mathit{Color} = \langle r{:}\texttt{int}, g{:}\texttt{int}, b{:}\texttt{int}\rangle \\ \texttt{type } \mathit{ColoredPoint} = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, c{:}\mathit{Color}\rangle \end{array}$$

Type definitions

- Instead, can also consider *defining new (named) types*

$$\texttt{deftype } T = \tau$$

- The term *generative* is sometimes used to refer to definitions that *create a new entity* rather than *introducing an abbreviation*
- Type abbreviations are usually not allowed to be recursive; type definitions can be.

$$\texttt{deftype } \mathit{IntList} = [\mathit{Nil} : \texttt{unit}, \mathit{Cons} : \texttt{int} \times \mathit{IntList}]$$

Type definitions vs. abbreviations in practice

- In Haskell, type abbreviations are introduced by `type`, while new types can be defined by `data` or `newtype` declarations.
- In Java, there is no explicit notation for type abbreviations; the only way to define a new type is to define a `class` or `interface`
- In Scala, type abbreviations are introduced by `type`, while the `class`, `object` and `trait` constructs define new types

Subtyping

- Suppose we have a function:

$$dist = \lambda p{:}Point.\ sqrt((p.x)^2 + (p.y)^2)$$

  for computing the distance to the origin.
- Only the x and y fields are needed for this, so we'd like to be able to use this on *ColoredPoint*s also.
- But, this doesn't typecheck:

$$dist(\langle x{=}8.0, y{=}12.0, c{=}purple\rangle) = 13.0$$

- We can introduce a *subtyping* relationship between *Point* and *ColoredPoint* to allow for this.

Subtyping

- Liskov proposed a guideline for subtyping:

Liskov Substitution Principle

If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.

- If we use $\tau <: \tau'$ to mean "τ is a subtype of τ'", and consider well-typedness to be desirable, then we can translate this to the following *subsumption* rule:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

- This says: if e has type τ_1 and $\tau_1 <: \tau_2$, then we can proceed by pretending it has type τ_2.

Record subtyping: width and depth

- There are several different ways to define subtyping for records.
- **Width subtyping:** subtype has same fields as supertype (with identical types), and may have additional fields at the end:

$$\frac{}{\langle l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_{n+k} : \tau_{n+k}\rangle <: \langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle}$$

- **Depth subtyping:** subtype's fields are pointwise subtypes of supertype

$$\frac{\tau_1 <: \tau_1' \quad \cdots \quad \tau_n <: \tau_n'}{\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle <: \langle l_1 : \tau_1', \ldots, l_n : \tau_n'\rangle}$$

- These rules can be combined. Optionally, field reordering can also be allowed (but is harder to implement).

Examples

- (We'll abbreviate $P = Point$, $P3 = Point3d$, $CP = ColoredPoint$ to save space...)
- So we have:

$$P3d = \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}, z{:}\mathtt{dbl}\rangle <: \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}\rangle = P$$

$$CP = \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}, c{:}Color\rangle <: \langle x{:}\mathtt{dbl}, y{:}\mathtt{dbl}\rangle = P$$

  but no other subtyping relationships hold
- So, we can call *dist* on *Point3d* or *ColoredPoint*:

$$\dfrac{\dfrac{x : P3d \vdash x : P3d \quad P3d <: P}{x : P3d \vdash x : P} \quad \dfrac{\vdots}{x : P3d \vdash dist : P \to \mathtt{dbl}}}{x : P3d \vdash dist(x) : \mathtt{dbl}}$$

Subtyping for pairs and variants

- For pairs, subtyping is componentwise

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

- Similarly for binary variants

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

- For named variants, can have additional subtyping rules (but this is rare)

Subtyping for functions

- When is $A_1 \to B_1 <: A_2 \to B_2$?
- Maybe componentwise, like pairs?

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

- But then we can do this (where $\Gamma(p) = P$):

$$\frac{\dfrac{\Gamma \vdash \lambda x.x : CP \to CP \quad \dfrac{CP <: P \quad CP <: CP}{CP \to CP <: P \to CP}}{\Gamma \vdash \lambda x.x : P \to CP} \quad \Gamma \vdash p : P}{\Gamma \vdash (\lambda x.x)p : CP}$$

- So, once *ColoredPoint* is a subtype of *Point*, we can change any *Point* to a *ColoredPoint* also. That doesn't seem right.

Covariant vs. contravariant

- For the result type of a function (and for pairs and other data structures), the direction of subtyping is preserved:

$$\frac{\tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1 \to \tau_2'}$$

- Subtyping of function results, pairs, etc., where order is preserved, is *covariant*.
- For the *argument* type of a function, the direction of subtyping is flipped:

$$\frac{\tau_1' <: \tau_1}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2}$$

- Subtyping of function arguments, where order is reversed, is called *contravariant*.

The "top" and "bottom" types

- `any`: a type that is a supertype of all types.
  - Such a type describes the common interface of all its subtypes (e.g. hashing, equality in Java)
  - In Scala, this is called `Any`
- `empty`: a type that is a subtype of all types.
  - Usually, such a type is considered to be *empty*: there cannot actually be any values of this type.
  - We've actually encountered this before, as the degenerate case of a choice type where there are zero chioces
  - In Scala, this type is called `Nothing`. So for any Scala type τ we have $Nothing <: \tau <: Any$.

Summary: Subtyping rules

$\boxed{\tau_1 <: \tau_2}$

$$\frac{}{\texttt{empty} <: \tau} \qquad \frac{}{\tau <: \texttt{any}} \qquad \frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Notice that we combine the covariant and contravariant rules for functions into a single rule.

Structural vs. Nominal subtyping

- The approach to subtyping considered so far is called *structural*.
- The names we use for type abbreviations don't matter, only their structure. For example, $Point3d <: Point$ because $Point3d$ has all of the fields of $Point$ (and more).
- Then $dist(p)$ also runs on $p : Point3d$ (and gives a nonsense answer!)
- So far, a defined type has no subtypes (other than itself).
- By default, definitions $ColoredPoint$, $Point$ and $Point3d$ are unrelated.

Structural vs. Nominal subtyping

- If we defined new types $Point'$ and $Point3d'$, rather than treating them as abbreviations, then we have more control over subtyping
- Then we can *declare* $ColoredPoint'$ to be a subtype of $Point'$

$\texttt{deftype}\ Point' = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}\rangle$
$\texttt{deftype}\ ColoredPoint' <: Point' = \langle x{:}\texttt{dbl}, y{:}\texttt{dbl}, c{:}Color\rangle$

- However, we could choose not to assert $Point3d'$ to be a subtype of $Point'$, preventing (mis)use of subtyping to view $Point3d'$s as $Point'$s.
- This *nominal* subtyping is used in Java and Scala
  - A defined type can only be a subtype of another if it is declared as such
  - More on this later!

Summary

- Today we covered:
  - Records, variants, and pattern matching
  - Type abbreviations and definitions
  - Subtyping
- Next time:
  - Polymorphism and type inference

Elements of Programming Languages

Lecture 8: Polymorphism and type inference

James Cheney

University of Edinburgh

October 21, 2016

Overview

- This week and next week, we will cover different forms of **abstraction**
  - type definitions, records, datatypes, subtyping
  - polymorphism, type inference
  - modules, interfaces
  - objects, classes
- Today:
  - polymorphism and type inference

Consider the humble identity function

- A function that returns its input:

```
def idInt(x: Int) = x
def idString(x: String) = x
def idPair(x: (Int,String)) = x
```

- Does the same thing no matter what the type is.
- But we cannot just write this:

```
def id(x) = x
```

(In Scala, every variable needs to have a type.)

Another example

- Consider a pair "swap" operation:

```
def swapInt(p: (Int,Int)) = (p._2,p._1)
def swapString(p: (String,String)) = (p._2,p._1)
def swapIntString(p: (Int,String)) = (p._2,p._1)
```

- Again, the code is the same in both cases; only the types differ.
- But we can't write

```
def swap(p) = (p._2,p._1)
```

What type should `p` have?

Another example

- Consider a higher-order function that calls its argument twice:

```
def twiceInt(f: Int => Int) = {x: Int => f(f(x))}
def twiceStr(f: String => String) =
   {x: String => f(f(x))}
```

- Again, the code is the same in both cases; only the types differ.
- But we can't write

```
def twice(f) = {x => f(f(x))}
```

What types should `f` and `x` have?

Type parameters

In Scala, function definitions can have *type parameters*

```
def id[A](x: A): A = x
```

This says: given a type `A`, the function `id[A]` takes an `A` and returns an `A`.

```
def swap[A,B](p: (A,B)): (B,A) = (p._2,p._1)
```

This says: given types `A,B`, the function `swap[A,B]` takes a pair `(A,B)` and returns a pair `(B,A)`.

```
def twice[A](f: A => A): A => A = {x:A => f(f(x))}
```

This says: given a type `A`, the function `twice[A]` takes a function `f: A => A` and returns a function of type `A => A`

Parametric Polymorphism

- Scala's type parameters are an example of a phenomenon called *polymorphism* (= "many shapes")
- More specifically, *parametric* polymorphism because the function is *parameterized* by the type.
    - Its behavior cannot "depend on" what type replaces parameter `A`.
    - The type parameter `A` is *abstract*
- We also sometimes refer to `A`, `B`, `C` etc. as *type variables*

Polymorphism: More examples

- Polymorphism is even more useful in combination with higher-order functions.
- Recall `compose` from the lab:

```
def compose[A,B,C](f: A => B, g: B => C) =
  {x:A => g(f(x))}
```

- Likewise, the `map` and `filter` functions:

```
def map[A,B](f: A => B, x: List[A]): List[B] = ...
def filter[A](f: A => Bool, x: List[A]): List[A] = ...
```

(though in Scala these are usually defined as methods of `List[A]` so the `A` type parameter and `x` variable are implicit)

Formalization

- We add *type variables* $A, B, C, \ldots$, *type abstractions*, *type applications*, and *polymorphic types*:

$$\begin{array}{rcl} e & ::= & \cdots \mid \Lambda A.\ e \mid e[\tau] \\ \tau & ::= & \cdots \mid A \mid \forall A.\ \tau \end{array}$$

- We also use (capture-avoiding) substitution of types for type variables in expressions and types.
- The type $\forall A.\ \tau$ is the type of expressions that can have type $\tau[\tau'/A]$ for any choice of A. (A is bound in τ.)
- The expression $\Lambda A.\ e$ introduces a type variable for use in e. (Thus, A is bound in any type annotations in e.)
- The expression $e[\tau]$ instantiates a type abstraction
- Define $\mathrm{L_{Poly}}$ to be the extension of $\mathrm{L_{Data}}$ with these features

Formalization: Type and type variables

- Complication: Types now have variables. What is their scope? When is a type variable in scope in a type?
- The polymorphic type $\forall A.\tau$ *binds* A in τ.
- We write $A \# \tau$ to say that type variable A is *fresh for* τ:

$$\frac{A \neq B}{A \# B} \qquad \frac{A \# \tau_1 \quad A \# \tau_2}{A \# \tau_1 \times \tau_2} \qquad \frac{A \# \tau_1 \quad A \# \tau_2}{A \# \tau_1 \to \tau_2}$$

$$\frac{A \# \tau_1 \quad A \# \tau_2}{A \# \tau_1 + \tau_2} \qquad \frac{}{A \# \forall A.\tau} \qquad \frac{A \neq B \quad A \# \tau}{A \# \forall B.\tau}$$

- $A \# x_1{:}\tau_1, \ldots, x_n{:}\tau_n \iff A \# \tau_1 \cdots A \# \tau_n$
- Alpha-equivalence and type substitution are defined similarly to expressions.

Formalization: Typechecking polymorphic expressions

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau \quad A \# \Gamma}{\Gamma \vdash \Lambda A.\ e : \forall A.\ \tau} \qquad \frac{\Gamma \vdash e : \forall A.\ \tau}{\Gamma \vdash e[\tau_0] : \tau[\tau_0/A]}$$

- Idea: $\Lambda A.\ e$ must typecheck with parameter A not already used elsewhere in type context
- $e[\tau_0]$ applies a polymorphic expression to a type. Result type obtained by substituting for A.
- The other rules are unchanged

Formalization: Semantics of polymorphic expressions

- To model evaluation, we add type abstraction as a possible value form:

$$v ::= \cdots \mid \Lambda A.e$$

- with rules similar to those for λ and application:

$\boxed{e \Downarrow v}$ for $\mathrm{L_{Poly}}$

$$\frac{e \Downarrow \Lambda A.\ e_0 \quad e_0[\tau/A] \Downarrow v}{e[\tau] \Downarrow v} \qquad \frac{}{\Lambda A.\ e \Downarrow \Lambda A.\ e}$$

- In $\mathrm{L_{Poly}}$, type information is irrelevant at run time.
- (Other languages, including Scala, do retain some run time type information.)

Convenient notation

- We can augment the syntactic sugar for function definitions to allow type parameters:

$$\texttt{let fun } f[A](x : \tau) = e \texttt{ in } \ldots$$

- This is equivalent to:

$$\texttt{let } f = \Lambda A.\ \lambda x : \tau.\ e \texttt{ in } \ldots$$

- In either case, a function call can be written as

$$f[\tau](x)$$

Examples in $\mathrm{L}_{\mathrm{Poly}}$

- Identity function

$$id = \Lambda A.\lambda x{:}A.\ x$$

- Swap

$$swap = \Lambda A.\Lambda B.\lambda x{:}A \times B.\ (\texttt{snd } x, \texttt{fst } x)$$

- Twice

$$twice = \Lambda A.\ \lambda f{:}A \to A.\lambda x{:}A.\ f(f(x))$$

- For example:

$$swap[\texttt{int}][\texttt{str}](1, "a") \Downarrow ("a", 1)$$

$$twice[\texttt{int}](\lambda x{:}\ 2 \times x)(2) \Downarrow 8$$

Examples, typechecked

$$\cfrac{\cfrac{\overline{x{:}A \vdash x{:}A}}{\vdash \lambda x{:}A.\ x : A \to A}}{\vdash \Lambda A.\lambda x{:}A.x : \forall A.A \to A}$$

$$\cfrac{\cfrac{\overline{\vdash swap : \forall A.\forall B.A \times B \to B \times A}}{\vdash swap[\texttt{int}] : \forall B.\texttt{int} \times B \to B \times \texttt{int}}}{\vdash swap[\texttt{int}][\texttt{str}] : \texttt{int} \times \texttt{str} \to \texttt{str} \times \texttt{int}}$$

Lists and parameterized types

- In Scala (and other languages such as Haskell and ML), type abbreviations and definitions can be *parameterized*.
- `List[_]` is an example: given a type `T`, it constructs another type `List[T]`

$$\texttt{deftype } List[A] = [Nil : \texttt{unit}; Cons : A \times List[A]]$$

- Such types are sometimes called *type constructors*
- (See tutorial questions on lists)
- We will revisit parameterized types when we cover modules

Other forms of polymorphism

- Polymorphism refers to several related techniques for "code reuse" or "overloading"
  - Subtype polymorphism: reuse based on inclusion relations between types.
  - Parametric polymorphism: abstraction over type parameters
  - Ad hoc polymorphism: Reuse of same name for multiple (potentially type-dependent) implementations (e.g. *overloading* + for addition on different numeric types, string concatenation etc.)
- These have some overlap
- We will discuss overloading, subtyping and polymorphism (and their interaction) in future lectures.

Type inference

- As seen in even small examples, specifying the type parameters of polymorphic functions quickly becomes tiresome

$$swap[\mathtt{int}][\mathtt{str}] \qquad map[\mathtt{int}][\mathtt{str}] \qquad \cdots$$

- Idea: Can we have the benefits of (polymorphic) typing, without the costs? (or at least: with fewer annotations)
- *Type inference*: Given a program without full type information (or with some missing), *infer* type annotations so that the program can be typechecked.

Hindley-Milner type inference

- A very influential approach was developed independently by J. Roger Hindley (in logic) and Robin Milner (in CS).
- Idea: Typecheck an expression symbolically, collecting "constraints" on the unknown type variables
- If the constraints have a common solution then this solution is a most general way to type the expression
  - Constraints can be solved using *unification*, an equation solving technique from automated reasoning/logic programming
- If not, then the expression has a type error

Hindley-Milner example [Non-examinable]

- As an example, consider *swap* defined as follows:

$$\vdash \lambda x : A.(\mathtt{snd}\ x, \mathtt{fst}\ x) : B$$

A, B are the as yet unknown types of x and *swap*.

Hindley-Milner example [Non-examinable]

- As an example, consider *swap* defined as follows:

$$\vdash \lambda x : A.(\texttt{snd}\ x, \texttt{fst}\ x) : B$$

  A, B are the as yet unknown types of x and *swap*.
- A lambda abstraction creates a function: hence $B = A \to A_1$ for some A_1 such that $x{:}A \vdash (\texttt{snd}\ x, \texttt{fst}\ x) : A_1$

Parametric Polymorphism Type inference

Hindley-Milner example [Non-examinable]

- As an example, consider *swap* defined as follows:

$$\vdash \lambda x : A.(\texttt{snd}\ x, \texttt{fst}\ x) : B$$

  A, B are the as yet unknown types of x and *swap*.
- A lambda abstraction creates a function: hence $B = A \to A_1$ for some A_1 such that $x{:}A \vdash (\texttt{snd}\ x, \texttt{fst}\ x) : A_1$
- A pair constructs a pair type: hence $A_1 = A_2 \times A_3$ where $x{:}A \vdash \texttt{snd}\ x : A_2$ and $x{:}A \vdash \texttt{fst}\ x : A_3$

Parametric Polymorphism Type inference

Hindley-Milner example [Non-examinable]

- As an example, consider *swap* defined as follows:

$$\vdash \lambda x : A.(\texttt{snd}\ x, \texttt{fst}\ x) : B$$

  A, B are the as yet unknown types of x and *swap*.
- A lambda abstraction creates a function: hence $B = A \to A_1$ for some A_1 such that $x{:}A \vdash (\texttt{snd}\ x, \texttt{fst}\ x) : A_1$
- A pair constructs a pair type: hence $A_1 = A_2 \times A_3$ where $x{:}A \vdash \texttt{snd}\ x : A_2$ and $x{:}A \vdash \texttt{fst}\ x : A_3$
- This can only be the case if $x : A_3 \times A_2$, i.e. $A = A_3 \times A_2$.

Parametric Polymorphism Type inference

Hindley-Milner example [Non-examinable]

- As an example, consider *swap* defined as follows:

$$\vdash \lambda x : A.(\texttt{snd}\ x, \texttt{fst}\ x) : B$$

  A, B are the as yet unknown types of x and *swap*.
- A lambda abstraction creates a function: hence $B = A \to A_1$ for some A_1 such that $x{:}A \vdash (\texttt{snd}\ x, \texttt{fst}\ x) : A_1$
- A pair constructs a pair type: hence $A_1 = A_2 \times A_3$ where $x{:}A \vdash \texttt{snd}\ x : A_2$ and $x{:}A \vdash \texttt{fst}\ x : A_3$
- This can only be the case if $x : A_3 \times A_2$, i.e. $A = A_3 \times A_2$.
- Solving the constraints: $A = A_3 \times A_2$, $A_1 = A_2 \times A_3$ and so $B = A_2 \times A_3 \to A_3 \times A_2$

Let-bound polymorphism [Non-examinable]

- An important additional idea was introduced in the ML programming language, to avoid the need to explicitly introduce type variables and apply polymorphic functions to type arguments
- When a function is defined using `let fun` (or `let rec`), first infer a type:

$$swap : A_2 \times A_3 \to A_3 \times A_2$$

- Then *abstract* over all of its free type parameters.

$$swap : \forall A.\forall B.A \times B \to B \times A$$

- Finally, when a polymorphic function is *applied*, infer the missing types.

$$swap(1, "a") \leadsto swap[\mathtt{int}][\mathtt{str}](1, "a")$$

ML-style inference: strengths and weaknesses

- Strengths
  - Elegant and effective
  - Requires no type annotations at all
- Weaknesses
  - Can be difficult to explain errors
  - In theory, can have exponential time complexity (in practice, it runs efficiently on real programs)
  - Very sensitive to extension: subtyping and other extensions to the type system tend to require giving up some nice properties
- (We are intentionally leaving out a lot of technical detail — HM type inference is covered in more detail in ITCS.)

Type inference in Scala

- Scala does not employ full HM type inference, but uses many of the same ideas.
- Type information in Scala flows from function arguments to their results

```
def f[A](x: List[A]): List[(A,A)] = ...
f(List(1,2,3)) // A must be Int, don't need f[Int]
```

- and sequentially through statement blocks

```
var l = List(1,2,3); // l: List[Int] inferred
var y = f(l); // y : List[(Int,Int)] inferred
```


Type inference in Scala

- Type information does **not** flow across arguments in the same argument list

```
def map[A](f: A => B, l: List[A]): List[B] = ...
scala> map({x: Int => x + 1}, List(1,2,3))
res0: List[Int] = List(2, 3, 4)
scala> map({x => x + 1}, List(1,2,3))
<console>:25: error: missing parameter type
```

- But it **can** flow from earlier argument lists to later ones:

```
def map2[A](l: List[A])(f: A => B): List[B] = ...
scala> map2(List(1,2,3)) {x => x + 1}
res1: List[Int] = List(2, 3, 4)
```

Type inference in Scala: strengths and limitations

- Compared to Java, many **fewer** annotations needed
- Compared to ML, Haskell, etc. many **more** annotations needed
- The reason has to do with Scala's integration of polymorphism and **subtyping**
  - needed for integration with Java-style object/class system
  - Combining subtyping and polymorphism is tricky (type inference can easily become undecidable)
  - Scala chooses to avoid global constraint-solving and instead propagate type information *locally*

Summary

- Today we covered:
  - The idea of thinking of the same code as having many different types
  - Parametric polymorphism: makes the type parameter explicit and abstract
  - Brief coverage of *type inference*.
- Next time:
  - Programs, modules, and interfaces

]

Elements of Programming Languages

Lecture 9: Programs, modules and interfaces

James Cheney

University of Edinburgh

October 25, 2016

Overview

- So far we have covered programming "in the small"
  - simple functional programming
  - abstractions: parametric polymorphism and subtyping
- Next few lectures: programming "in the large"
- Today
  - "Programs" as collections of definitions
  - Namespace management — *packages*
  - Abstract data types — *modules* and *interfaces*
- We will mostly work "by example" using Scala — formalizing modules, interfaces involves a lot of bureaucracy.

Programs

- What is a program?
  - In L_{Poly}, a program is an expression; any functions defined in L_{Poly} are local to the expression

$$\begin{array}{l} \texttt{let fun } f(x:\tau) = e_1 \texttt{ in} \\ \texttt{let fun } g(y:\tau') = e_2 \texttt{ in} \\ \vdots \\ e \end{array}$$

- Scope management is easier with these simplistic forms, but isn't very modular
- In particular, we can't easily split a program up into parts that do unrelated work.

Declarations and Programs

- Most languages support *declarations*

$$\begin{array}{rcl} Decl \ni d & ::= & \texttt{let } x = e; \mid \texttt{let fun } f(y:\tau) = e; \\ & \mid & \texttt{let rec } f(y:\tau):\tau' = e; \\ & \mid & \texttt{type } T = \tau; \mid \texttt{deftype } T = \tau; \end{array}$$

- A *program* is a sequence of declarations. The names x, f, T are in scope in the subsequent declarations.
  - Variation: In some languages (Haskell, Scala), the order of declarations within a program is unimportant, and names can be referenced before they are used.
  - Variation: In some languages, only certain "top-level" declarations are allowed (e.g. classes/interfaces in Java)

Entry points

- The *entry point* is the place where execution starts when the program is run

```
public static void main(String[] args) {...}
```

- Can be specified in different ways:
  - Executable: specify a particular function that is called first (e.g. `main` in C/C++, Java, Scala)
  - Scripting: entry point is start of program, expressions or statements run in order
  - Web applications: entry points are functions such as `doGet`, `doPost` in Java's `Servlet` interface
  - Reactive: provide *callbacks* to handle one or more *events* (e.g. JavaScript handlers for mouse actions)

Programming in the large

- What is the largest program you've written (or maintained)?
  - 1000 lines — 1 file?
  - 10,000 lines? 10 files?
  - 100,000 lines? 100 files?
- Sooner or later, someone is going to want to use the same name for different things.
- If there are n programmers, then there are $O(n^2)$ possible sources of name conflicts.
- *Namespaces* provide a way to compartmentalize names to avoid ambiguity.

Example: Packages in Java

```
// com/widget/round/Widget.java
package com.widget.round
class Widget {...
}

// com/widget/square/Widget.java
package com.widget.square
class Widget { ...
}
```

- We can reuse `Widget` and disambiguate: `com.widget.square.Widget` vs. `com.widget.round.Widget`
- (Package names track the directory hierarchy in Java.)

Importing

- Given a namespace, we can *import* it

```
import com.widget.round.Widget
```

  - This brings a *single* name defined in a namespace into the current scope

```
import com.widget.round.*
```

  - This brings *all* names defined in a namespace into the current scope
- In Java, importing can only happen at the top level of a file, and imported names are always classes or interfaces.
  - (Scala is more flexible, as we'll see)

Code reuse and abstract data types

- Another important concern for programming in the large is *code reuse*.
- We'd like to implement (or reuse) certain key data structures once and for all, in a *modular* way
  - Examples: Lists, stacks, queues, sets, maps, etc.
- An *abstract data type* (ADT) is a type together with some operations on it
  - Abstract means the type definition (and operation implementations) are not visible to the rest of the program
  - Only the types of the operations are visible (the *interface*)
  - An ADT also has a *specification* describing its behavior

Running example: priority queues in Scala

Using Scala `objects`, here is an initial priority queue ADT:

```
object PQueue {
  type T = ...
  val empty: T
  def insert(n: Int,pq: T): T
  def remove(pq:T): (Int,T)
}
```

- (Similar to Java `class` with only static members)
- Specification:
  - A priority queue represents a set of integers.
  - `empty` corresponds to the empty set
  - `insert` adds to the set
  - `remove` removes the *least* element of the set

Implementing priority queues

- One implementation: sorted lists (others possible)

```
object ListPQueue {
  type T = List[Int]
  val empty: T = Nil
  def insert(n: Int,pq: T): T = pq match {
    case Nil => List(n)
    case x::xs =>
      if (n < x) {n::pq} else {x::insert(n,xs)}
  }
  def remove(pq:T) = pq match {
    case x::xs => (x,xs) // otherwise error
  }
}
```


Importing

- Like packages, `objects` provide a form of namespace

```
object ListPQueue {
  ...
}
val pq = ListPQueue.insert(1,ListPQueue.empty)
import ListPQueue._
val pq2 = remove(pq)
```

- Importing can be done inside other scopes (unlike Java)

```
def singleton(x: Int) {
  import ListPQueue._
  insert(x,empty)
}
```

ListPQueue isn't abstract

- If we only use the ListPQueue operations, the specification is satisfied
- However, the ListPQueue.T type allows non-sorted lists
- So we can violate the specification by passing remove a non-sorted list!

```
remove(List(2,1))
// returns 2, should return 1
```

- This violates the (implicit) invariant that ListPQueue.T is a sorted list.
- So, users of this module need to be more careful to use it correctly.

Programs Namespaces and Packages Modules and Interfaces

One solution (?)

- As in Java, we can make some components private

```
object ListPQueue {
  private type T = List[Int]
  private val foo: T = List(1)
}
```

- This stops us from accessing foo

```
scala> ListPQueue.foo
<console>:20: error: (foo cannot be accessed)
```

- However, T is still visible as List[Int]!

```
scala> ListPQueue.remove(List(2,1))
res10: (Int, List[Int]) = (2,List(1))
```

Programs Namespaces and Packages Modules and Interfaces

Interfaces

- Another way to hide information about the implementation of a module is to specify an *interface*
- (This may be familiar from Java already. Haskell type classes also can act as interfaces.)
- We'd like to use an interface PQueue that says there is some type T with operations:

```
empty: T
insert: (Int,T) => T
remove: T => (Int,T)
```

but prevent clients from knowing (or relying on) the definition of T.

Programs Namespaces and Packages Modules and Interfaces

Traits in Scala

- Scala doesn't exactly have Java-like interfaces, but its traits can play a similar role.

```
trait PQueue {
  type T = List[Int]
  val empty: T
  def insert(n: Int, pq: T): T
  def remove(pq: T): (Int,T)
}
```

- (We'll say more about why Scala uses the terms object and trait instead of module and interface later...)

Implementing an interface

- Already, the trait interface hides information about the implementations of the operations. But, now we can go further and hide the definition of `T`!

```
trait PQueue {
  type T // abstract!
}
```

- Now we can specify that `ListPQueue` *implements* `PQueue` using the `extends` keyword:

```
    object ListPQueue extends PQueue {...}
```

- This assertion needs be *checked* to ensure that all of the components of `PQueue` are present and have the right types!

Checking a module against an interface

```
trait PQueue {
  type T
  val empty: T
  def insert(n: Int, pq: T): T
  def remove(pq: T): (Int,T)
}
```

- An implementation needs to define `T` to be some type τ
- It needs to provide a value `empty:` τ
- It needs to provide functions `insert` and `remove` with the corresponding types (replacing `T` with τ)
- If any are missing or types don't match, error.
- (Note: this is related to type inference, and there can be similar complications!)

Interfaces allow multiple implementations

- We can now provide other implementations of `PQueue`

```
object ListPQueue extends PQueue {...}
object SetPQueue extends PQueue {...}
```

- Also, in Scala, `objects` can be passed as values, and `extends` implies a subtyping relationship
- So, we can write a function that uses any implementation of `PQueue`, and run it with different implementations:

```
def make(m: PQueue) =
  m.insert(42,m.insert(17,m.empty))
scala> make(ListPQueue)
```


Data abstraction

- Even though `ListPQueue` satisfies the `PQueue` interface, its definition of `T = List[Int]` is still visible
- However, `T` is *abstract* to clients that use the `PQueue` interface
- So, we can't do this:

```
scala> def bad(m: PQueue) = m.remove(List(2,1))
<console>:18: error: type mismatch;
 found   : List[Int]
 required: m.T
       def bad(m: PQueue) = m.remove(List(2,1))
```

Implementing multiple interfaces

- An interface gives a "view" of a module (possibly hiding some details).
- Modules can also satisfy more than one interface.

```
trait HasSize {
  type T
  def size(x: T): Int
}
object ListPQueue extends PQueue with HasSize {
  ...
  def size(pq: T) = pq.length
}
```

- (This is slightly hacky, since it relies on using the same type name `T` as `PQueue` uses. We'll revisit this later.)

Representation independence

- If we have two implementations of the same interface, how do we know they are providing "equivalent" behavior?
- *Representation independence* means that the clients of the interface can't distinguish the two implementations using the operations of the interface
  - (even if their actual run time behavior is very different)
- This is much easier in a strongly typed language because the abstraction barrier is enforced by type system
- In other languages, client code needs to be more careful to avoid depending on (or violating) intended abstraction barriers

Modules and interfaces, in general

$$
\begin{array}{rcl}
\mathit{Decl} \ni d & ::= & \texttt{let}\ x = e; \mid \texttt{let fun}\ f(x:\tau) = e; \\
& \mid & \texttt{let rec}\ f(x:\tau):\tau' = e; \\
& \mid & \texttt{type}\ T = \tau; \mid \texttt{deftype}\ T = \tau; \\
& \mid & \texttt{module}\ M\ \{d_1 \cdots d_n\} \mid \texttt{import}\ q \\
& \mid & \texttt{interface}\ S\ \{s_1 \cdots s_n\} \\
\mathit{Spec} \ni s & ::= & \texttt{val}\ x:\ \tau; \mid \texttt{type}\ T; \mid \texttt{type}\ T = \tau; \\
\mathit{QName} \ni q & ::= & x \mid M.q \mid S.q \mid \_
\end{array}
$$

This a simplified form of the (influential) Standard ML module language. (We aren't going to formalize the details.)
Note: Allows arbitrary nesting of modules, interfaces
Not shown: need to allow qualified names in code also

Summary

- As programs grow in size, we want to:
  - split programs into components (*packages* or *modules*)
  - use package or module scope and structured names to refer to components
  - use interfaces to hide implementation details from other parts of the program
- We've given a high-level idea of how these components fit together, illustrated using Scala
- Next time:
  - Object-oriented constructs (objects, classes)

Elements of Programming Languages

Lecture 10: Objects and Classes

James Cheney

University of Edinburgh

October 28, 2016

Overview

- Last time: "programming in the large"
  - Programs, packages/namespaces, importing
  - Modules and interfaces
  - Mostly: using Scala for examples
- Today: the elephant in the room:
  - Objects and Classes
  - A taste of "advanced" OOP constructs: inner classes, anonymous objects and mixins
  - Illustrate using examples in Scala, and some comparisons with Java

Objects

- An *object* is a module with some additional properties:
  - **Encapsulation**: Access to an object's components can be limited to the object itself (or to a subset of objects)
  - **Self-reference**: An object is a value and its methods can refer to the object's fields and methods (via an implicit parameter, often called `this` or `self`)
  - **Inheritance**: An object can inherit behavior from another "parent" object
- Objects/inheritance are tied to *classes* in some (but not all) OO languages
- In Scala, the `object` keyword creates a *singleton object* ("class with only one instance")
- (in Java, objects can only be created as instances of *classes*)

Self-Reference

- Inside an object definition, the `this` keyword refers to the object being defined.
- This provides another form of recursion:

```
object Fact {
  def fact (n: Int): Int = {
    if (n == 0) {1} else {n * this.fact(n-1)}
  }
}
```

- Moreover, as we'll see, the recursion is *open*: the method that is called by `this.foo(x)` depends on what `this` is at run time.

Encapsulation and Scope

- An object can place restrictions on the *scope* of its members
- Typically used to prevent *external interference* with 'internal state' of object
- For example: Java, C++, C# all support
  - `private` keyword: "only visible to this object"
  - `public` keyword: "visible to all"
- Java: package scope (default): visible only to other components in the same package
- Scala: `private[X]` allows *qualified* scope: "private to (class/object/trait/package) `X`"
- Python, Javascript: don't have (enforced) `private` scope (relies on programmer goodwill)

Classes

- A *class* is an interface with some additional properties:
  - **Instantiation**: classes can describe how to construct associated objects (*instances* of the class)
  - **Inheritance**: classes may *inherit* from zero or more *parent* classes as well as *implement* zero or more interfaces
  - **Abstraction**: Classes may be *abstract*, that is, may name but not define some fields or methods
  - **Dynamic dispatch**: The choice of which method is called is determined by the run-time type of a class instance, not the static type available at the call
- Not all object-oriented languages have classes!
  - Smalltalk, JavaScript are well-known exceptions
  - Such languages nevertheless often use *prototypes*, or commonly-used objects that play a similar role to classes

Constructing instances

- Classes typically define special functions that create new instances, called *constructors*
  - In C++/Java, constructors are defined explicitly and separately from the initialized data
  - In Scala, there is usually one "default" constructor whose parameters are in scope in the whole class body
  - (additional constructors can be defined as needed)
- Constructors called with the `new` keyword

```
class C(x: Int, y: String) {
  val i = x
  val s = y
  def this(x: Int) = this(x,"default")
}
scala> val c1 = new C(1,"abc")
scala> val c2 = new C(1)
```


Inheritance

- An object can *inherit* from another.
- This means: the parent object, and its components, become "part of" the child object
  - accessible using `super` keyword
  - (though some components may not be directly accessible)
- In Java (and Scala), a class extends exactly one superclass (`Object`, if not otherwise specified)
- In C++, a class can have **multiple** superclasses
- Non-class-based languages, such as JavaScript and Smalltalk, support inheritance directly on objects via *extension*

Subtyping

- As (briefly) mentioned last week, an object `Obj` that extends a trait `Tr` is automatically a *subtype* (`Obj <: Tr`)
- Likewise, a class `Cl` that extends a trait `Tr` is a subtype of `Tr` (`Cl <: Tr`)
- A class (or object) `Sub` that extends another class `Super` is a subtype of `Super` (`Sub <: Super`)
- However, subtyping and inheritance are *distinct* features:
  - As we've already seen, subtyping can exist without inheritance
  - moreover, subtyping is about *types*, whereas inheritance is about *behavior* (code)

Inheritance and encapsulation

- Inheritance complicates the picture for encapsulation somewhat.
- `private` keyword prevents access from outside the class (including any subclasses).
- `protected` keyword means "visible to instances of this object and its subclasses"
- Scala: Both `private` and `protected` can be qualified with a scope `[X]` where `X` is a package, class or object.

```
class A { private[A] val a = 1
          protected[A] val b = 2 }
class B extends A {
  def foo() = a + b
} // "a" not found
```


Cross-instance sharing

- Classes in Java can have *static* fields/members that are shared across all instances
- Static methods can access `private` fields and methods
- `static` is also allowed in interfaces (but only as of Java 8)
- Class with only static members $\sim$ module
- C++: `friend` keyword allows sharing between classes on a case-by-case basis

Companion Objects

- Scala has no `static` keyword
- Scala instead uses *companion objects*
  - Companion = object with the same name as the class and defined in the same scope
  - Companions can access each others' `private` components

```
object Count { private var x = 1 }
class Count { def incr() {Count.x = Count.x+1} }
```

- Note: This can only be done in compiled code, not interactively

Multiple inheritance and the *diamond problem*

- As noted, C++ allows *multiple inheritance*
- Suppose we did this (in Scala terms):

```
class Win(val x: Int, val y: Int)
class TextWin(...) extends Win
class GraphicsWin(...) extends Win
class TextGraphicsWin(...)
  extends TextWin and GraphicsWin
```

- In C++, this means there are two copies of `Win` inside `TextGraphicsWin`
- They can easily become out of sync, causing problems
- Multiple inheritance is also difficult to implement (efficiently); many languages now avoid it

Abstraction

- A class may leave some components undefined
  - Such classes must be marked `abstract` in Java, C++ and Scala
  - To instantiate an abstract class, must provide definitions for the methods (e.g. in a subclass)
- Abstract classes can define common behavior to be inherited by subclasses
- In Scala, abstract classes can also have unknown *type* components
  - (optionally with subtype constraints)

```
abstract class ConstantVal {
  type T <: AnyVal
  val c: T
} // a constant of any value type
```


Dynamic dispatch

- An abstract method can be implemented in different ways by different subclasses
- When an abstract method is called on an instance, the corresponding implementation is determined by the *run-time type* of the instance.
- (necessarily in this case, since the abstract class provides no implementation)

```
abstract class A { def foo(): String}
class B extends A { def foo() = "B"}
class C extends A { def foo() = "C" }
scala> val b:A = new B
scala> val c:A = new C
scala> (b.foo(), c.foo())
```


Overriding

- An inherited method that is already defined by a superclass can be *overridden* in a subclass
- This means that the subclass's version is called on that subclass's instances using dynamic dispatch
- In Java, `@Override` annotation is optional, checked documentation that a method overrides an inherited method
- In Scala, must use `override` keyword to clarify intention to override a method

```
class A { def foo() = "A"}
class B extends A { override def foo() = "B" }
scala> val b: A = new B
scala> b.foo()
class C extends A { def foo() = "C" } // error
```

Type tests and coercions

- Given `x: A`, Java/Scala allow us to *test* whether its run-time type is actually subclass `B`

```
scala> b.isInstanceOf[B]
```

- and to *coerce* such a reference to `y: B`

```
scala> val b2: B = b.asInstanceOf[B]
```

- Warning: these features can be used to violate type abstraction!

```
def weird[A](x: A) = if (x.isInstanceOf[Int]) {
  (x.asInstanceOf[Int]+1).asInstanceOf[A]
  } else {x}
```


Advanced constructs

- So far, we've covered the "basic" OOP model (circa Java 1.0)
- Modern languages extend this in several ways
- We can define a class/object inside another class:
  - As a member of the enclosing class (tied to a specific instance)
  - or as a static member (shared across all instances)
  - As a local definition inside a method
  - As an anonymous local definition
- Some languages also support *mixins* (e.g. Scala traits)
- Scala supports similar, somewhat more uniform composition of classes, objects, and traits

Classes/objects as members

- In Scala, classes and objects (and traits) can be nested arbitrarily

```
class A { object B { var x = 1 } }
scala> val a = new A

object C {class D { var x = 1 } }
scala> val d = new C.D

class E { class F { var x = 1 } }
scala> val e = new E
scala> val f = new e.F
```


Summary

- Today
  - Objects, encapsulation, self-reference
  - Classes, inheritance, abstraction, dynamic dispatch
- This is only the tip of a very large iceberg...
  - there are almost as many "object-oriented" programming models as languages
  - the design space, and "right" formalisms, are still active areas of research
- Next time:
  - Inner classes, anonymous objects, mixins, parameterized types
  - Combining object-oriented and functional programming

Elements of Programming Languages

Lecture 11: Object-oriented functional programming

James Cheney

University of Edinburgh

November 1, 2016

Overview

- We've now covered:
  - basics of functional programming (with semantics)
  - basics of modular and OO programming (via Scala examples)
- Today, finish discussion of "programming in the large":
  - some more advanced OO constructs
  - and how they co-exist with/support functional programming in Scala
  - *list comprehensions* as an extended example

Advanced constructs

- So far, we've covered the "basic" OOP model (circa Java 1.0), plus some Scala-isms
- Modern languages extend this model in several ways
- We can define a structure (class/object/trait) inside another:
  - As a member of the enclosing class (tied to a specific instance)
  - or as a static member (shared across all instances)
  - As a local definition inside a method
  - As an anonymous local definition
- Java (since 1.5) and Scala support "generics" (*parameterized types* as well as polymorphic functions)
- Some languages also support *mixins* (e.g. Scala traits)

Motivating inner class example

- A nested/inner class has access to the private/protected members of the containing class
- So, we can use nested classes to expose an interface associated with a specific object:

```
class List<A> {
  private A head;
  private List<A> tail;
  class ListIterator<A> implements Iterator<A> {
    ... (can access head, tail)
  }
}
```

Classes/objects as members

- In Scala, classes and objects (and traits) can be nested arbitrarily

```
class A { object B { val x = 1 } }
scala> val a = new A

object C {class D { val x = 1 } }
scala> val d = new C.D

class E { class F { val x = 1 } }
scala> val e = new E
scala> val f = new e.F
```


Local classes

- A *local class* (Java terminology) is a class that is defined inside a method

```
def foo(): Int = {
  val z = 1
  class X { val x = z + 1}
  return (new X).x
}
scala> foo()
res0: Int = 2
```


Anonymous classes/objects

- Given an interface or parent class, we can define an anonymous instance without giving it an explicit name
- In Java, called an *anonymous local class*
- In Scala, looks like this:

```
abstract class Foo { def foo() : Int }
val foo1 = new Foo { def foo() = 42 }
```

- We can also give a *local name* to the instance (useful since `this` may be shadowed)

```
val foo2 = new Foo { self =>
  val x = 42
  def foo() = self.x
}
```


Parameterized types

- As mentioned earlier, types can take *parameters*
- For example, `List[A]` has a type parameter `A`
- This is related to (but different from) polymorphism
  - A polymorphic function (like `map`) has a type that is parameterized by a given type.
  - A parameterized type (like `List[_]`) is a type *constructor*: for every type `T`, it constructs a type `List[T]`.

Defining parameterized types

- In Scala, there are basically three ways to define parameterized types:
  - In a type abbreviation (NB: multiple parameters)

```
type Pair[A,B] = (A,B)
```

  - in a (abstract) class definition

```
abstract class List[A]
case class Cons[A](head: A, tail: List[A])
    extends List[A]
```

  - in a trait definition

```
trait Stack[A] { ...
}
```


Using parameterized types inside a structure

- The type parameters of a structure are implicitly available to all components of the structure.
- Thus, in the `List[A]` class, `map`, `flatMap`, `filter` are declared as follows:

```
abstract class List[A] {
  ...
  def map[B](f: A => B): List[B]
  def filter(p: A => Boolean): List[A]
  def flatMap[B](f: A => List[B]): List[B]
    // applies f to each element of this,
    // and concatenates results
}
```


Parameterized types and subtyping

- By default, a type parameter is *invariant*
  - That is, neither covariant nor contravariant
- To indicate that a type parameter is *covariant*, we can prefix it with +

```
abstract class List[+A] // see tutorial 6
```

- To indicate that a type parameter is *contravariant*, we can prefix it with -

```
trait Fun[-A,+B] // see next few slides...
```

- Scala **checks** to make sure these variance annotations make sense!

Type bounds

- Type parameters can be given *subtyping bounds*
- For example, in an interface (that is, trait or abstract class) `I`:

```
type T <: C
```

  says that abstract type member `T` is constrained to be a subtype of `C`.
- This is checked for any module implementing `I`
- Similarly, type parameters to function definitions, or class/trait definitions, can be bounded:

```
fun f[A <: C](...) = ...
class D[A <: C] { ... }
```

- Upper bounds `A >: U` are also possible...

Traits as mixins

- So far we have used Scala's `trait` keyword for "interfaces" (which can include type members, unlike Java)
- However, traits are considerably more powerful:
  - Traits can contain fields
  - Traits can provide ("default") method implementations
- This means traits provide a powerful form of modularity: *mixin composition*
  - Idea: a trait can specify extra fields and methods providing a "behavior"
  - Multiple traits can be "mixed in"; most recent definition "wins" (avoiding some problems of multipel inheritance)
- Java 8's support for "default" methods in interfaces also allows a form of mixin composition.

Tastes great, and look at that shine!

- Shimmer is a floor wax!

```
trait FloorWax { def clean(f: Floor) { ... } }
```

- No, it's a delicious dessert topping!

```
trait TastyDessertTopping {
  val calories = 1000
  def addTo(d: Dessert) { d.addCal(calories) }
}
```

- In Scala, it can be both:

```
object Shimmer extends FloorWax
                 with TastyDessertTopping { ... }
```


Pay no attention to the man behind the curtain...

- Scala bills itself as a "multi-paradigm" or "object-oriented, functional" language
- How do the "paradigms" actually fit together?
- Some features, such as case classes, are more obviously "object-oriented" versions of "functional" constructs
- Until now, we have pretended pairs, λ-abstractions, etc. are primitives in Scala
- **They are not primitives**; and they need to be implemented in a way compatible with Java/JVM assumptions
  - But how do they really work?

Function types as interfaces

- Suppose we define the following interface:

```
trait Fun[-A,+B] { // A contravariant, B covariant
  def apply(x: A): B
}
```

- This says: an object implementing `Fun[A,B]` has an `apply` method
- Note: This is basically the `Function` trait in the Scala standard library!
  - Scala translates `f(x)` to `f.apply(x)`
  - Also, `{x: T => e}` is essentially syntactic sugar for `new Function[Int,Int] {def apply(x:T) = e }`!

Iterators and collections in Java

- Java provides standard interfaces for *iterators* and *collections*

```
interface Iterator<E> {
  boolean hasNext()
  E next()
  ...
}
interface Collection<E> {
  Iterator<E> iterator()
  ...
}
```

- These allow programming over different types of collections in a more abstract way than "indexed for loop"

Iterators and `foreach` loops

- Since Java 1.5, one can write the following:

```
for(Element x : coll) {
  ... do stuff with x ...
}
```

  Provided `coll` implements the `Collection<Element>` interface
- This is essentially syntactic sugar for:

```
for(Iterator<Element> i = coll.iterator();
    i.hasNext(); ) {
  Element x = i.next();
  ... do stuff with x ...
}
```


`foreach` in Scala

- Scala has a similar `for` construct (with slightly different syntax)

```
for (x <- coll) { ... do something with x ... }
```

- For example:

```
scala> for (x <- List(1,2,3)) { println(x) }
1
2
3
```


`foreach` in Scala

- The construct `for (x <- coll) { e }` is syntactic sugar for:

```
coll.foreach{x => ... do something with x ...}
```

  if `x: T` and `coll` has method `foreach: (A => ()) => ()`
- Scala expands `for` loops **before** checking that `coll` actually provides `foreach` of appropriate type
- If not, you get a somewhat mysterious error message...

```
scala> for (x <- 42) {println(x)}
<console>:11: error: value foreach is not a
  member of Int
```

Comprehensions: Mapping

- Scala (in common with Haskell, Python, C#, F# and others) supports a rich "comprehension syntax"
- Example:

```
scala> for(x <- List("a","b","c")) yield (x + "z")
res0: List[Int] = List(az,bz,cz)
```

- This is shorthand for:

```
List("a","b","c").map{x => x + "z"}
```

where `map[B](f: A => B): List[B]` is a method of `List[A]`.

- (In fact, this works for any object implementing such a method.)

Comprehensions: Filtering

- Comprehensions can also include *filters*

```
scala> for(x <- List("a","b","c");
          if (x != "b")) yield (x + "z")
res0: List[Int] = List(az,cz)
```

- This is shorthand for:

```
List("a","b","c").filter{x => x != "b"}
  .map{x => x + "z"}
```

where `filter(f: A => Boolean): List[A]` is a method of `List[A]`.

Comprehensions: Multiple Generators

- Comprehensions can also iterate over several lists

```
scala> for(x <- List("a","b","c");
           y <- List("a","b","c");
           if (x != y)) yield (x + y)
res0: List[Int] = List(ab,ac,ba,bc,ca,cb)
```

- This is shorthand for:

```
List("a","b","c").flatMap{x =>
  List("a","b","c").flatMap{y =>
    if (x != y) List(x + y) else {Nil}}}
```

where `flatMap(f: A => List[B]): List[B]` is a method of `List[A]`.

Summary

- In the last few lectures we've covered
  - Modules and interfaces
  - Objects and classes
  - How they interact with subtyping, type abstraction
  - and how they can be used to implement "functional" features (particularly in Scala)
- This concludes our tour of "programming in the large"
- (though there is much more that could be said)
- Next time:
  - imperative programming

Elements of Programming Languages

Lecture 12: Imperative programming

James Cheney

University of Edinburgh

November 4, 2016

The story so far

- So far we've mostly considered *pure* computations.
- Once a variable is bound to a value, the value *never changes*.
  - that is, variables are *immutable*.
- This is **not** how most programming languages treat variables!
  - In most languages, we can *assign* new values to variables: that is, variables are *mutable* by default
- Just a few languages are completely "pure" (Haskell).
- Others strike a balance:
  - e.g. Scala distinguishes immutable (`val`) variables and mutable (`var`) variables
  - similarly `const` in Java, C

Mutable vs. immutable

- Advantages of immutability:
  - Referential transparency (substitution of equals for equals); programs easier to reason about and optimize
  - Types tell us more about what a program can/cannot do
- Advantages of mutability:
  - Some common data structures easier to implement
  - Easier to translate to machine code (in a performance-preserving way)
  - Seems closely tied to popular OOP model of "objects with hidden state and public methods"
- Today we'll consider programming with assignable variables and loops (L_{While}) and then discuss procedures and other forms of control flow

While-programs

- Let's start with a simple example: L_{While}, with *statements*

$$\begin{array}{rcl} \mathit{Stmt} \ni s & ::= & \mathtt{skip} \mid s_1; s_2 \mid x := e \\ & \mid & \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \mid \mathtt{while}\ e\ \mathtt{do}\ s \end{array}$$

- `skip` does nothing
- $s_1; s_2$ does s_1, then s_2
- $x := e$ evaluates e and **assigns** the value to x
- $\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2$ evaluates e, and evaluates s_1 or s_2 based on the result.
- $\mathtt{while}\ e\ \mathtt{do}\ s$ tests e. If true, evaluate s and **loop**; otherwise stop.
- We typically use {} to parenthesize statements.

A simple example: factorial again

- In Scala, mutable variables can be defined with `var`

```
var n = ...
var x = 1
while(n > 0) {
  x = n * x
  n = n-1
}
```

- In L_{While}, all variables are mutable

$$x := 1; \texttt{while}\ (n > 0)\ \texttt{do}\ \{x := n * x; n := n - 1\}$$

An interpreter for L_{While}

We will define a *pure* interpreter:

```
def exec(env: Env[Value], s: Stmt): Env[Value] =
s match {
  case Skip => env
  case Seq(s1,s2) =>
    val env1 = exec(env, s1)
    exec(env1,s2)
  case IfThenElseS(e,s1,s2) => eval(env,e) match {
    case BoolV(true) => exec(env,s1)
    case BoolV(false) => exec(env,s2)
   }
  ...
}
```


An interpreter for L_{While}

```
def exec(env: Env[Value], s: Stmt): Env[Value] =
s match {
  ...
  case WhileDo(e,s) => eval(env, e) match {
    case BoolV(true) =>
      val env1 = exec(env,s)
      exec(env1, WhileDo(e,s))
    case BoolV(false) => env
  }
  case Assign(x,e) =>
    val v = eval(env,e)
    env + (x -> v)
}
```


While-programs: evaluation

$\boxed{\sigma, s \Downarrow \sigma'}$

$$\frac{}{\sigma, \texttt{skip} \Downarrow \sigma} \qquad \frac{\sigma, s_1 \Downarrow \sigma' \quad \sigma', s_2 \Downarrow \sigma''}{\sigma, s_1; s_2 \Downarrow \sigma''}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s_1 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma'} \qquad \frac{\sigma, e) \Downarrow \texttt{false} \quad \sigma, s_2 \Downarrow \sigma'}{\sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \sigma'}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, s \Downarrow \sigma' \quad \sigma', \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma''}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma''}$$

$$\frac{\sigma, e \Downarrow \texttt{false}}{\sigma, \texttt{while}\ e\ \texttt{do}\ s \Downarrow \sigma} \qquad \frac{\sigma, e \Downarrow v}{\sigma, x := e \Downarrow \sigma[x := v]}$$

- Here, we use evaluation in context $\sigma, e \Downarrow v$ (cf. Assignment 2)

Examples

- $x := y + 1; z := 2 * x$

$$\dfrac{\dfrac{\sigma_1, y+1 \Downarrow 2}{\sigma_1, x := y+1 \Downarrow \sigma_2} \quad \dfrac{\sigma_2, 2*x \Downarrow 4}{\sigma_2, z := 2*x \Downarrow \sigma_3}}{\sigma_1, x := y+1; z := 2*x \Downarrow \sigma_3}$$

- where

$$\begin{aligned} \sigma_1 &= [y := 1] \\ \sigma_2 &= [x := 2, y := 1] \\ \sigma_3 &= [x := 2, y := 1, z := 4] \end{aligned}$$

Other control flow constructs

- We've taken "if" (with both "then" and "else" branches) and "while" to be primitive
- We can **define** some other operations in terms of these:

$$\begin{array}{rcl} \texttt{if } e \texttt{ then } s & \iff & \texttt{if } e \texttt{ then } s \texttt{ else skip} \\ \texttt{do } s \texttt{ while } e & \iff & s; \texttt{while } e \texttt{ do } s \\ \texttt{for } (i \in n \ldots m) \texttt{ do } s & \iff & i := n; \\ & & \texttt{while } i \leq m \texttt{ do } \{ \\ & & \quad s; i = i + 1 \\ & & \} \end{array}$$

- as seen in C, Java, etc.

Procedures

- L_{While} is not a realistic language.
- Among other things, it lacks *procedures*
- Example (C/Java):

```
int fact(int n) {
  int x = 1;
  while(n > 0) {
    x = x*n;
    n = n-1;
  }
  return x;
}
```

- Procedures can be added to L_{While} (much like functions in L_{Rec})
- Rather than do this, we'll show how to combine L_{While} with L_{Rec} later.

Structured vs. unstructured programming [Non-examinable]

- All of the languages we've seen so far are *structured*
  - meaning, control flow is managed using `if`, `while`, procedures, functions, etc.
- However, low-level machine code doesn't have any of these.
- A machine-code program is just a sequence of instructions in memory
- The only control flow is branching:
  - "unconditionally go to instruction at address *n*"
  - "if some condition holds, go to instruction at address *n*"
- Similarly, "goto" statements were the main form of control flow in many early languages

"GO TO" Considered Harmful [Non-examinable]

- In a famous letter (CACM 1968), Dijkstra listed many disadvantages of "goto" and related constructs
- It allows you to write "spaghetti code", where control flow is very difficult to decipher
- For efficiency/historical reasons, many languages include such "unstructured" features:
  - "goto" — jump to a specific program location
  - "switch" statements
  - "break" and "continue" in loops
- It's important to know about these features, their pitfalls and their safe uses.

`goto` in C [Non-examinable]

- The C (and C++) language includes `goto`
- In C, `goto L` jumps to the statement labeled `L`
- A typical (relatively sane) use of `goto`

```
... do some stuff ...
  if (error) goto error;
... do some more stuff ...
  if (error2) goto error;
... do some more stuff...
error: .. handle the error...
```

- We'll see other, better-structured ways to do this using exceptions.

`goto` in C: pitfalls [Non-examinable]

- The scope of the `goto L` statement and the target `L` might be different
- for that matter, they might not even be in the same procedure!
- For example, what does this do:

```
goto L;
if(1) {
    int k = fact(3);
L:  printf("%d",k);
}
```

- Answer: `k` will be some random value!

`goto`: caveats [Non-examinable]

- `goto` can be used safely in C, but is best avoided unless you have a really good reason
- e.g. very high performance/systems code
- Safe use: within same procedure/scope
- Or: to jump "out" of a nested loop

goto fail [Non-examinable]

- What's wrong with this picture?

```
if (error test 1)
  goto fail;
if (error test 2)
  goto fail;
  goto fail;
if (error test 3)
  goto fail;
...
fail:  ... handle error ...
```

- (In C, braces on `if` are optional; if they're left out, only the first `goto fail` statement is conditional!)
- This led to an Apple SSL security vulnerability in 2014 (see https://gotofail.com/)

switch statements [Non-examinable]

- We've seen `case` or `match` constructs in Scala
- The `switch` statement in C, Java, etc. is similar:

```
switch (month) {
  case 1: print("January"); break;
  case 2: print("February"); break;
  ...
  default: print("unknown month"); break;
}
```

- However, typically the argument must be a base type like `int`

switch statements: gotchas [Non-examinable]

- See the `break;` statement?
- It's an important part of the control flow!
  - it says "now jump out the end of the switch statement"

```
month = 1;
switch (month) {
  case 1: print("January");
  case 2: print("February");
  ...
  default: print("unknown month");
} // prints all months!
```

- Can you think of a good reason why you would want to leave out the `break`?

Break and continue [Non-examinable]

- The `break` and `continue` statements are also allowed in loops in C/Java family languages.

```
for(i = 0; i < 10; i++) {
  if (i % 2 == 0) continue;
  if (i == 7) break;
  print(i);
}
```

- "Continue" says *Skip the rest of this iteration of the loop.*
- "Break" says *Jump to the next statement after this loop*

Break and continue [Non-examinable]

- The `break` and `continue` statements are also allowed in loops in C/Java family languages.

```
for(i = 0; i < 10; i++) {
  if (i % 2 == 0) continue;
  if (i == 7) break;
  print(i);
}
```

- "Continue" says *Skip the rest of this iteration of the loop*.
- "Break" says *Jump to the next statement after this loop*
- This will print `135` and then exit the loop.

Labeled break and continue [Non-examinable]

- In Java, `break` and `continue` can use labels.

```
OUTER: for(i = 0; i < 10; i++) {
  INNER:  for(j = 0; j < 10; j++) {
             if (j > i) continue INNER;
             if (i == 4) break OUTER;
             print(j);
  }
}
```

- This will print `001012` and then exit the loop.

Labeled break and continue [Non-examinable]

- In Java, `break` and `continue` can use labels.

```
OUTER: for(i = 0; i < 10; i++) {
  INNER:  for(j = 0; j < 10; j++) {
             if (j > i) continue INNER;
             if (i == 4) break OUTER;
             print(j);
  }
}
```

- This will print `001012` and then exit the loop.
- (Labeled) break and continue accommodate some of the safe uses of `goto` without as many sharp edges

Summary

- Many real-world programming languages have:
  1. mutable state
  2. structured control flow (if/then, while, exceptions)
  3. procedures
- We've showed how to model and interpret L_{While}, a simple imperative language
- and discussed a variety of (unstructured) control flow structures, such as "goto", "switch" and "break/continue".
- Next time:
  - Small-step semantics and type soundness

Elements of Programming Languages

Lecture 13: Small-step semantics and type safety

James Cheney

University of Edinburgh

November 8, 2016

Overview

- For the remaining lectures we consider some *cross-cutting* considerations for programming language design.
  - Last time: Imperative programming
- Today:
  - Finer-grained (small-step) evaluation
  - Type safety

Refresher

- In the first 6 lectures we covered:
  - Basic arithmetic (L_{Arith})
  - Conditionals and booleans (L_{If})
  - Variables and let-binding (L_{Let})
  - Functions and recursion (L_{Rec})
  - Data structures (L_{Data})
- formalized using big-step evaluation ($e \Downarrow v$) and type judgments ($\Gamma \vdash e : \tau$)
- and implemented using Scala interpreters

Limitations of big-step semantics

- Big-step semantics is convenient, but also limited
- It says how to evaluate the "whole program" (expression) to its "final value"
- *But what if there is no final value?*
  - Expressions like $1 + \texttt{true}$ simply don't evaluate
  - Nonterminating programs don't evaluate either, but for a different reason!
- As we will see in later lectures, it is also difficult to deal with other features, like exceptions, using big-step semantics

Small-step semantics

- We will now consider an alternative: *small-step semantics*

$$e \mapsto e'$$

- which says how to evaluate an expression "one step at a time"
- If $e_0 \mapsto \cdots \mapsto e_n$ then we write $e_0 \mapsto^* e_n$. (in particular, for $n = 0$ we have $e_0 \mapsto^* e_0$)
- We want it to be the case that $e \mapsto^* v$ if and only if $e \Downarrow v$.
- But $\mapsto$ provides more detail about how this happens.
- It also allows expressions to "go wrong" (get stuck before reaching a value)

Small-step semantics: $\mathrm{L}_{\mathrm{Arith}}$

$e \mapsto e'$ for $\mathrm{L}_{\mathrm{Arith}}$

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1 \oplus e_2 \mapsto v_1 \oplus e_2'}$$

$$\frac{}{v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{v_1 \times v_2 \mapsto v_1 \times_{\mathbb{N}} v_2}$$

- If the first subexpression of $\oplus$ can take a step, apply it
- If the first subexpression is a value and the second can take a step, apply it
- If both sides are values, perform the operation
- Example:

$$1 + (2 \times 3) \mapsto 1 + 6 \mapsto 7$$

Small-step semantics: L_{If}

$e \mapsto e'$ for L_{If}

$$\frac{}{v == v \mapsto \mathtt{true}} \qquad \frac{v_1 \neq v_2}{v_1 == v_2 \mapsto \mathtt{false}}$$

$$\frac{e \mapsto e'}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto \mathtt{if}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}$$

$$\frac{}{\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_1}$$

$$\frac{}{\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_2}$$

- If the conditional test is not a value, evaluate it one step
- Otherwise, evaluate the corresponding branch

$$\begin{aligned} \mathtt{if}\ 1 == 2\ \mathtt{then}\ 3\ \mathtt{else}\ 4 &\mapsto \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ 3\ \mathtt{else}\ 4 \\ &\mapsto 4 \end{aligned}$$

Small-step semantics: $\mathrm{L}_{\mathrm{Let}}$

$e \mapsto e'$ for $\mathrm{L}_{\mathrm{Let}}$

$$\frac{e_1 \mapsto e_1'}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mapsto \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2}$$

$$\frac{}{\mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 \mapsto e_2[v_1/x]}$$

- If the expression e_1 is not yet a value, evaluate it one step
- Otherwise, substitute it and proceed
- Example:

$$\begin{aligned} \mathtt{let}\ x = 1 + 1\ \mathtt{in}\ x \times x &\mapsto \mathtt{let}\ x = 2\ \mathtt{in}\ x \times x \\ &\mapsto 2 \times 2 \\ &\mapsto 4 \end{aligned}$$

Small-step semantics: $\mathrm{L}_{\mathrm{Lam}}$

$\boxed{e \mapsto e'}$ for $\mathrm{L}_{\mathrm{Lam}}$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}$$

$$\frac{}{(\lambda x.\ e)\ v \mapsto e[v/x]}$$

- If the function part is not a value, evaluate it one step
- If the function is a value and the argument isn't, evaluate it one step
- If both function and argument are values, substitute and proceed

$$\begin{aligned} ((\lambda x.\lambda y.x + y)\ 1)\ 2 &\mapsto (\lambda y.1 + y)\ 2 \\ &\mapsto 1 + 2 \mapsto 3 \end{aligned}$$

Small-step semantics: $\mathrm{L}_{\mathrm{Rec}}$

$\boxed{e \mapsto e'}$ for $\mathrm{L}_{\mathrm{Rec}}$

$$\frac{}{(\texttt{rec}\ f(x).\ e)\ v \mapsto e[\texttt{rec}\ f(x).e/f, v/x]}$$

- Same rules for evaluation inside application
- Note that we need to substitute $\texttt{rec}\ f(x).e$ for f.
- Suppose *fact* is the factorial function:

$$\begin{aligned} \mathit{fact}\ 2 &\mapsto \texttt{if}\ 2 == 0\ \texttt{then}\ 1\ \texttt{else}\ 2 \times \mathit{fact}(2-1) \\ &\mapsto \texttt{if false then}\ 1\ \texttt{else}\ 2 \times \mathit{fact}(2-1) \\ &\mapsto 2 \times \mathit{fact}(2-1) \mapsto 2 \times \mathit{fact}(1) \\ &\mapsto 2 \times (\texttt{if}\ 1 == 0\ \texttt{then}\ 1\ \texttt{else}\ 1 \times \mathit{fact}(1-1)) \\ &\mapsto 2 \times (\texttt{if false then}\ 1\ \texttt{else}\ 1 \times \mathit{fact}(1-1)) \\ &\mapsto 2 \times (1 \times \mathit{fact}(1-1)) \mapsto 2 \times (1 \times \mathit{fact}(0)) \\ &\mapsto^* 2 \times (1 \times 1) \mapsto 2 \times 1 \mapsto 2 \end{aligned}$$

Judgments and Rules, in general

- A *judgment* is a relation among one or more abstract syntax trees.
- Examples so far: $e \Downarrow v$, $\Gamma \vdash e : \tau$, $e \mapsto e'$
- We have been defining judgments using *rules* of the form:

$$\frac{}{Q} \qquad \frac{P_1 \quad \cdots \quad P_n}{Q}$$

- where $P_1, \ldots, P_n$ and Q are judgments.

Meaning of Rules

- A rule of the form:

$$\frac{}{Q}$$

  is called an *axiom*. It says that Q is always derivable.
- A rule of the form

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

  says that judgment Q is derivable if $P_1, \ldots, P_n$ are derivable.
- Symbols like e, v, τ in rules stand for arbitrary expressions, values, or types.
- (If you have taken Logic Programming: These rules are a lot like Prolog clauses!)

Rule induction

Induction on derivations of $e \Downarrow v$

Suppose $P(-,-)$ is a predicate over pairs of expressions and values. If:

- $P(v,v)$ holds for all values v
- If $P(e_1,v_1)$ and $P(e_2,v_2)$ then $P(e_1+e_2, v_1 +_{\mathbb{N}} v_2)$
- If $P(e_1,v_1)$ and $P(e_2,v_2)$ then $P(e_1 \times e_2, v_1 \times_{\mathbb{N}} v_2)$

then $e \Downarrow v$ implies $P(e,v)$.

- Rule induction can be derived from mathematical induction on the size (or height) of the derivation tree.
- (Much like structural induction.)
- We won't formally prove this.

Example: $e \Downarrow v$ implies $e \mapsto^* v$

- As an example, we'll show a few cases of the forward direction of:

Theorem (Equivalence of big-step and small-step evaluation)

$e \Downarrow v$ if and only if $e \mapsto^* v$.

Base case.

If the derivation is of the form

$$\frac{}{n \Downarrow n}$$

for some number n, then $e = n$ is already a value $v = n$, so no steps are needed to evaluate it, i.e. $n \mapsto^* n$ in zero steps. □

Example: $e \Downarrow v$ implies $e \mapsto^* v$

Inductive case.

If the derivation is of the form

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2}$$

then by induction, we know $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$. Using the small-step rules, we can then show

$$e_1 + e_2 \mapsto^* v_1 + e_2 \mapsto^* v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2$$

□

- The case for $\times$ is similar.

Type soundness

- The central property of a type system is *soundness*.
- Roughly speaking, soundness means "well-typed programs don't go wrong" [Milner].
- But what exactly does "go wrong" mean?
  - For large-step: hard to say
  - For small-step: "go wrong" means "stuck" expression e that is not a value and cannot take a step.
- We could show something like:

Theorem (Soundness)

If $\vdash e : \tau$ and $e \mapsto^* v$ then $\vdash v : \tau$.

- This says that if an expression evaluates to a value, then the value has the right type.

Type soundness revisited

- We can decompose soundness into two parts:

Lemma (Progress)

If $\vdash e : \tau$ then either e is a value or for some e' we have $e \mapsto e'$.

Lemma (Preservation)

If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$

- Combining these two, can show:

Theorem (Soundness)

If $\vdash e : \tau$ and $e \mapsto^* v$ then $\vdash v : \tau$.

- We will *sketch* these properties for L_{If} (leaving out a lot of formal detail)

Progress for L_{If}

Progress is proved by induction on $\vdash e : \tau$ derivations. We show some representative cases.

Progress for +.

$$\frac{\vdash e_1 : \mathtt{int} \quad e_2 : \mathtt{int}}{\vdash e_1 + e_2 : \mathtt{int}}$$

If the derivation is of the above form, then by induction e_1 is either a value or can take a step, and likewise for e_2. There are three cases.

- If $e_1 \mapsto e_1'$ then $e_1 + e_2 \mapsto e_1' + e_2$.
- If e_1 is a value v_1 and $e_2 \mapsto e_2'$, then $v_1 + e_2 \mapsto v_1 + e_2'$.
- If both e_1 and e_2 are values then they must both be numbers $n_1, n_2 \in \mathbb{N}$, so $e_1 + e_2 \mapsto n_1 +_{\mathbb{N}} n_2$.

$\square$

Progress for L_{If}

Progress for `if`.

If the derivation is of the form

$$\frac{\vdash e : \mathtt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

then by induction, either e is a value or can take a step. There are two cases:

- If $e \mapsto e'$ then $\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto \mathtt{if}\ e'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$.
- If e is a value, it must be either `true` or `false`. Then either $\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_1$ or $\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mapsto e_2$.

$\square$

Preservation for L_{If}

Preservation is proved by induction on the structure of $\vdash e : \tau$. We'll consider some representative cases:

Preservation for +.

$$\frac{\vdash e_1 : \mathtt{int} \quad \vdash e_2 : \mathtt{int}}{\vdash e_1 + e_2 : \mathtt{int}}$$

If the derivation is of the above form, there are three cases.

- If $e_i = v_i$ and $v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2$ then obviously $\vdash v_1 +_{\mathbb{N}} v_2 : \mathtt{int}$.
- If $e_1 + e_2 \mapsto e_1' + e_2$ where $e_1 \mapsto e_1'$, then since $\vdash e_1 : \mathtt{int}$, we have $\vdash e_1' : \mathtt{int}$, so $\vdash e_1' + e_2 : \mathtt{int}$ also.
- The case where $e_1 = v_1$ and $v_1 + e_2 \mapsto v_1 + e_2'$ is similar.

$\square$

Preservation for L_{If}

Preservation for `if`.

If the derivation is of the form

$$\frac{\vdash e : \texttt{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

then there are three cases:

- If $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ where $e \mapsto e'$, then by induction we can show that $\vdash e' : \texttt{bool}$ and $\vdash \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau$.
- If $e = \texttt{true}$ then $\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mapsto e_1$, so we already know $\vdash e_1 : \tau$.
- The case for $\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \mapsto e_2$ is similar.

$\square$

Type soundness for L_{Let} [non-examinable]

- Progress: straightforward (a "let" can always take a step)
- Preservation: Suppose we have

$$\frac{\vdash v_1 : \tau' \quad x{:}\tau' \vdash e_2 : \tau}{\vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 : \tau} \qquad \frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ e_2 \mapsto e_2[v_1/x]}$$

  We need to show that $\vdash e_2[v_1/x] : \tau$
- For this we need a *substitution lemma*

Lemma (Substitution)

If $\Gamma, x{:}\tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash e[e'/x] : \tau$

Type soundness for L_{Rec} [non-examinable]

- Progress: If an application term is well-formed:

$$\frac{\vdash e_1 : \tau_1 \to \tau_2 \quad \vdash e_2 : \tau_1}{\vdash e_1\ e_2 : \tau_2}$$

  then by induction, e_1 is either a value or $e_1 \mapsto e_1'$ for some e_1'. If it is a value, it must be either a lambda-expression or a recursive function, so $e_1\ e_2$ can take a step. Otherwise, $e_1\ e_2 \mapsto e_1'\ e_2$.
- Preservation: Similar to `let`, using substitution lemma for the cases

$$\begin{array}{rcl} (\lambda x.\ e)\ v & \mapsto & e[v/x] \\ (\texttt{rec}\ f(x).\ e)\ v & \mapsto & e[\texttt{rec}\ f(x).\ e/f, v/x] \end{array}$$

Summary

- Today we have presented
  - Small-step evaluation: a finer-grained semantics
  - Induction on derivations
  - Type soundness (details for L_{If})
  - Sketch of type soundness for L_{Rec} [Non-examinable]
- Deep breath: No more proofs from now on.
- Remaining lectures cover cross-cutting language features, which often have subtle interactions with each other
  - Next time: Guest lecture by Michel Steuwer on *DSLs and rewrite-based optimizations for performance-portable parallel programming*

Elements of Programming Languages

Lecture 14: References, Arrays, and Resources

James Cheney

University of Edinburgh

November 15, 2016

Overview

- Over the final few lectures we are exploring *cross-cutting* design issues
- Today we consider a way to incorporate mutable variables/assignment into a functional setting:
  - References
  - Interaction with subtyping and polymorphism
  - Resources, more generally

References

- In L_{While}, all variables are **mutable** and **global**
- This makes programming fairly tedious and it's easy to make mistakes
- There's also no way to create new variables (short of coming up with a new variable name)
- Can we smoothly add mutable state side-effects to L_{Poly}?
- Can we provide imperative features within a mostly-functional language?

References

- Consider the following language L_{Ref} extending L_{Poly}:

$$\begin{array}{rcl} e & ::= & \cdots \mid \mathtt{ref}(e) \mid\ !e \mid e_1 := e_2 \mid e_1; e_2 \\ \tau & ::= & \cdots \mid \mathtt{ref}[\tau] \end{array}$$

- Idea: $\mathtt{ref}(e)$ evaluates e to v and creates a **new reference cell** containing v
- $!e$ evaluates e to a reference and **looks up its value**
- $e_1 := e_2$ evaluates e_1 to a reference cell and e_2 to a value and **assigns** the value to the reference cell.
- $e_1; e_2$ evaluates e_1, ignores value, then evaluates e_2

References: Types

$\boxed{\Gamma \vdash e : \tau}$ for L_{Ref}

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ref}(e) : \mathtt{ref}[\tau]} \qquad \frac{\Gamma \vdash e : \mathtt{ref}[\tau]}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{ref}[\tau] \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathtt{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau}$$

- $\mathtt{ref}(e)$ creates a reference of type τ if $e : \tau$
- $!e$ gets a value of type τ if $e : \mathtt{ref}[\tau]$
- $e_1 := e_2$ updates reference $e_1 : \mathtt{ref}[\tau]$ with value $e_2 : \tau$. Its return value is ().
- $e_1; e_2$ evaluates e_1, ignores the resulting value, and evaluates e_2.

References in Scala

Recall that `var` in Scala makes a variable mutable:

```
class Ref[A](val x: A) {
  private var a = x
  def get = a
  def set(y: A) = { a = y }
}
scala> val x = new Ref[Int](1)
x: Ref[Int] = Ref@725bef66
scala> x.get
res3: Int = 1
scala> x.set(12)
scala> x.get
res5: Int = 12
```


Interpreting references in Scala using Ref

```
case class Ref(e: Expr) extends Expr
case class Deref(e: Expr) extends Expr
case class Assign(e: Expr, e2: Expr) extends Expr
case class Cell(l: Ref[Value]) extends Value

def eval(env: Env[Value], e: Expr) = e match { ...
  case Ref(e)        => Cell(new Ref(eval(env,e)))
  case Deref(e)      => eval(env,e) match {
    case Cell(r) => r.get
  }
  case Assign(e1,e2) => eval(env,e1) match {
    case Cell(r) => r.set(eval(env,e2))
  }
} // Note: This isn't how Assignment 3 does it!
```


Imperative Programming and Procedures

- Once we add references to a functional language (e.g. L_{Poly}), we can use function definitions and lambda-abstraction to define *procedures*
- Basically, a procedure is just a function with return type `unit`

```
val x = new Ref(42)
def incrBy(n: Int): () = {
  x.set(x.get + n)
}
```

- Such a procedure does not return a value, and is only executed for its "side effects" on references
- Using the same idea, we can embed all of the constructs of L_{While} in L_{Ref} (see tutorial)

References: Semantics

- Small steps $\sigma, e \mapsto \sigma', e'$, where $\sigma : Loc \to Value$. "in initial state σ, expression e can step to e' with state σ'."
- What does $\texttt{ref}(e)$ evaluate to? A *pointer* or *memory cell location*, $\ell \in Loc$

$$v \quad ::= \quad \cdots \mid \ell$$

- These special values only appear during evaluation.

$\sigma, e \mapsto \sigma', e'$ for L_{Ref}

$$\frac{\ell \notin locs(\sigma)}{\sigma, \texttt{ref}(v) \mapsto \sigma[\ell := v], \ell}$$

$$\frac{}{\sigma, !\ell \mapsto \sigma, \sigma(\ell)} \qquad \frac{}{\sigma, \ell := v \mapsto \sigma[\ell := v], ()}$$

References: Semantics

- We also need to change all of the existing small-step rules to pass σ through...

$\sigma, e \mapsto \sigma', e'$

$$\frac{\sigma, e_1 \mapsto \sigma', e_1'}{\sigma, e_1 \oplus e_2 \mapsto \sigma', e_1' \oplus e_2} \qquad \frac{\sigma, e_2 \mapsto \sigma', e_2'}{\sigma, v_1 \oplus e_2 \mapsto \sigma', v_1 \oplus e_2'}$$

$$\frac{}{\sigma, v_1 + v_2 \mapsto \sigma, v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{\sigma, v_1 \times v_2 \mapsto \sigma, v_1 \times_{\mathbb{N}} v_2}$$

$$\vdots$$

- Subexpressions may contain references (leading to allocation or updates), so we need to allow σ to change in any subexpression evaluation step.

References: Semantics

- Finally, we need rules that evaluate inside the reference constructs themselves:

$\sigma, e \mapsto \sigma', e'$

$$\frac{\sigma, e \mapsto \sigma', e'}{\sigma, \texttt{ref}(e) \mapsto \sigma', \texttt{ref}(e')} \qquad \frac{\sigma, e \mapsto \sigma', e'}{\sigma, !e \mapsto \sigma', !e'}$$

$$\frac{\sigma, e_1 \mapsto \sigma', e_1'}{\sigma, e_1 := e_2 \mapsto \sigma', e_1' := e_2} \qquad \frac{\sigma, e_2 \mapsto \sigma', e_2'}{\sigma, v_1 := e_2 \mapsto \sigma', v_1 := e_2'}$$

- Notice again that we need to allow for updates to σ.
- For example, to evaluate $\texttt{ref}(\texttt{ref}(42))$

References: Examples

- Simple example

$$\begin{array}{ll} & \texttt{let}\ r = \texttt{ref}(42)\ \texttt{in}\ r := 17; !r \\ \mapsto & [\ell := 42], \texttt{let}\ r = \ell\ \texttt{in}\ r := 17; !r \\ \mapsto & [\ell := 42], \ell := 17; !\ell \\ \mapsto & [\ell := 17], !\ell \mapsto [\ell := 17], 17 \end{array}$$

References: Examples

- Simple example

$$\begin{array}{ll} & \mathtt{let}\ r = \mathtt{ref}(42)\ \mathtt{in}\ r := 17; !r \\ \mapsto & [\ell := 42], \mathtt{let}\ r = \ell\ \mathtt{in}\ r := 17; !r \\ \mapsto & [\ell := 42], \ell := 17; !\ell \\ \mapsto & [\ell := 17], !\ell \mapsto [\ell := 17], 17 \end{array}$$

- Aliasing/copying

$$\begin{array}{ll} & \mathtt{let}\ r = \mathtt{ref}(42)\ \mathtt{in}\ (\lambda x.\lambda y.x := !y + 1)\ r\ r \\ \mapsto & [\ell = 42], \mathtt{let}\ r = \ell\ \mathtt{in}\ (\lambda x.\lambda y.x := !y + 1)\ r\ r \\ \mapsto & [\ell = 42], (\lambda x.\lambda y.x := !y + 1)\ \ell\ \ell \\ \mapsto & [\ell = 42], (\lambda y.!\ell := y + 1)\ \ell \\ \mapsto & [\ell = 42], \ell := !\ell + 1 \mapsto [\ell = 42], \ell := 42 + 1 \\ \mapsto & [\ell = 42], \ell := 43 \mapsto [\ell = 43], () \end{array}$$

References Semantics of references Resources

Something's missing

- We didn't give a rule for $e_1; e_2$. It's pretty straightforward (exercise!)
- actually, $e_1; e_2$ is *definable* as

$$e_1; e_2 \iff \mathtt{let}\ \_ = e_1\ \mathtt{in}\ e_2$$

where $\_$ stands for any variable not already in use in e_1, e_2.
- Why?
  - To evaluate $e_1; e_2$, we evaluate e_1 for its side effects, ignore the result, and then evaluate e_2 for its value (plus any side effects)
  - Evaluating $\mathtt{let}\ \_ = e_1\ \mathtt{in}\ e_2$ first evaluates e_1, then binds the resulting value to some variable not used in e_2, and finally evaluates e_2.

References Semantics of references Resources

Reference semantics: observations

- Notice that any subexpression can create, read or assign a reference:

$$\mathtt{let}\ r = \mathtt{ref}(1)\ \mathtt{in}\ (r := 1000; 3) + !r$$

- This means that evaluation order really matters!
- Do we get 4 or 1003 from the above?
  - With left-to-right order, $r := 1000$ is evaluated first, then $!r$, so we get 1003
  - If we evaluated right-to-left, then $!r$ would evaluate to 1, before assigning $r := 1000$, so we would get 4
- However, the small-step rules clarify that existing constructs evaluate "as usual", with no side-effects.

References Semantics of references Resources

Arrays

- Arrays generalize references to allow getting and setting by *index* (i.e. a reference is a one-element array)

$$\begin{array}{lcl} e & ::= & \cdots \mid \mathtt{array}(e_1, e_2) \mid e_1[e_2] \mid e_1[e_2] := e_3 \\ \tau & ::= & \cdots \mid \mathtt{array}[\tau] \end{array}$$

- $\mathtt{array}(n, \mathit{init})$ creates an array of n elements, initialized to *init*
- $\mathit{arr}[i]$ gets the ith element; $\mathit{arr}[i] := v$ sets the ith element to v
- This introduces the potential problem of *out-of-bounds accesses*
- Typing, evaluation rules for arrays: exercise

References and subtyping

- Consider `Integer <: Object`, `String <: Object`
- Suppose we allowed *contravariant* subtyping for `Ref`, i.e. `Ref[-A]`
- which is obviously silly: we shouldn't expect a reference to `Object` to be castable to `String`.
- We could then do the following:

```
val x: Ref[Object] = new Ref(new Integer(42))
// String <: Object,
//   hence Ref[Object] <: Ref[String]
x.get.length // unsound!
```


References and subtyping

- Consider `Int <: Object`, `String <: Object`
- Suppose we allowed *covariant* subtyping for `Ref`, i.e. `Ref[+A]`
- We could then do the following:

```
val x: Ref[String] = new Ref(new String("asdf"))
def bad(y: Ref[Object]) = y.set(new Integer(42))
bad(x) // x still has type Ref[String]!
x.get.length() // unsound!
```

- Therefore, mutable parameterized types like `Ref` must be *invariant* (neither covariant nor contravariant)
- (Java got this wrong, for built-in array types!)

References and polymorphism [non-examinable]

- A related problem: references can violate type soundness in a language with Hindley-Milner style type inference and let-bound polymorphism (e.g. ML, OCaml, F#)

```
let r = ref (fn x => x) in
r := (fn x => x + 1);
!r(true)
```

- `r` initially gets inferred type $\forall A.A \to A$
- We then assign r to be a function of type `int` $\to$ `int`
- and then apply r to a boolean!
- Accepted solution: the *value restriction* - the right-hand side of a polymorphic `let` must be a value.
- (e.g., in Scala, polymorphism is only introduced via function definitions)

Resources

- References, arrays illustrate a common *resource* pattern:
  - Memory cells (references, arrays, etc.)
  - Files/file handles
  - Database, network connections
  - Locks
- Usage pattern: allocate/open/acquire, use, deallocate/close/release
- Key issues:
  - How to ensure proper use?
  - How to ensure eventual deallocation?
  - How to avoid attempted use after deallocation?

Design choices regarding references and pointers

- Some languages (notably C/C++) distinguish between type τ and type $\tau*$ ("pointer to τ"), i.e. a mutable reference
- Other languages, notably Java, consider many types (e.g. classes) to be "reference types", i.e., all variables of that type are really mutable (and nullable!) references.
- In Scala, variables introduced by `val` are immutable, while using `var` they can be assigned.
- In Haskell, as a pure, functional language, all variables are immutable; references and mutable state are available but must be handled specially

Safe allocation and use of resources

- In a strongly typed language, we can ensure safe resource use by ensuring all expressions of type `ref`$[\tau]$ are properly *initialized*
- C/C++ does **not** do this: a pointer $\tau*$ may be "uninitialized" (not point to an allocated τ block). Must be initialized separately via `malloc` or other operations.
- Java (sort of) does this: an expression of reference type τ is a reference to an allocated τ (or null!)
- Scala, Haskell don't allow "silent" null values, and so a τ is always an allocated structure
- Moreover, a `ref`$[\tau]$ is always a reference to an allocated, mutable τ

Safe deallocation of resources?

- Unfortunately, types are not as helpful in enforcing safe deallocation.
- One problem: forgetting to deallocate (*resource leaks*). Leads to poor performance or run-time failure if resources exhausted.
- Another problem: deallocating the same resource more than once (*double free*), or trying to use it after it's been deallocated
- A major reason is *aliasing*: copies of references to allocated resources can propagate to unpredictable parts of the program
- *Substructural typing* discipline (cf. guest lecture) can help with this, but remains an active research topic...

Main approaches to deallocation

- C/C++: explicit deallocation (`free`) must be done by the programmer.
  - (This is very very hard to get right.)
- Java, Scala, Haskell use *garbage collection*. It is the runtime's job to decide when it is safe to deallocate resources.
  - This makes life much easier for the programmer, but requires a much more sophisticated implementation, and complicates optimization/performance tuning
- Lexical scoping or exception handling works well for ensuring deallocation in certain common cases (e.g. files, locks, connections)
- Other approaches include reference counting, regions, etc.

Summary

- We continued to explore design considerations that affect many aspects of a language
- Today:
  - references and mutability, in generality
  - interaction with subtyping and polymorphism
  - some observations about other forms of resources and the "allocate/use/deallocate" pattern

Elements of Programming Languages

Lecture 15: Evaluation strategies and laziness

James Cheney

University of Edinburgh

November 18, 2016

Overview

- Final few lectures: cross-cutting language design issues
- So far:
  - Type safety
  - References, arrays, resources
- Today:
  - Evaluation strategies (by-value, by-name, by-need)
  - Impact on language design (particularly handling *effects*)

Evaluation order

- We've noted already that some aspects of small-step semantics seem arbitrary
  - For example, left-to-right or right-to-left evaluation
- Consider the rules for $+, \times$. There are two kinds: *computational* rules that actually do something:

$$\frac{}{v_1 + v_2 \mapsto v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{v_1 \times v_2 \mapsto v_1 \times_{\mathbb{N}} v_2}$$

- and *administrative* rules that say how to evaluate inside subexpressions:

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1 \oplus e_2 \mapsto v_1 \oplus e_2'}$$

Evaluation order

- We can vary the *evaluation order* by changing the administrative rules.
- To evaluate right-to-left:

$$\frac{e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'} \qquad \frac{e_1 \mapsto e_1'}{e_1 \oplus v_2 \mapsto e_1' \oplus v_2}$$

- To leave the evaluation order *unspecified*:

$$\frac{e_1 \mapsto e_1'}{e_1 \oplus e_2 \mapsto e_1' \oplus e_2} \qquad \frac{e_2 \mapsto e_2'}{e_1 \oplus e_2 \mapsto e_1 \oplus e_2'}$$

by lifting the constraint that the other side has to be a value.

Call-by-value

- So far, function calls evaluate arguments to values *before* binding them to variables

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'} \qquad \frac{}{(\lambda x.\ e)\ v \mapsto e[v/x]}$$

- This *evaluation strategy* is called *call-by-value*.
  - Sometimes also called *strict* or *eager*
- "Call-by-value" historically refers to the fact that expressions are evaluated before being passed as parameters
- It is the default in most languages

Example

- Consider $(\lambda x.x \times x)\ (1 + 2 \times 3)$
- Then we can derive:

$$\frac{\dfrac{2 \times 3 \mapsto 6}{1 + 2 \times 3 \mapsto 1 + 6}}{(\lambda x.x \times x)\ (1 + 2 \times 3) \mapsto (\lambda x.x \times x)\ (1 + 6)}$$

- Next:

$$\frac{1 + 6 \mapsto 7}{(\lambda x.x \times x)\ (1 + 6) \mapsto (\lambda x.x \times x)\ 7}$$

- Finally:

$$\frac{}{(\lambda x.x \times x)\ 7 \mapsto 7 \times 7 \mapsto 49}$$

Interpreting call-by-value

We evaluate subexpressions fully before substituting them for variables:

```
  def eval (e: Expr): Value = e match {
  ...
  case Let(x,e1,e2) => eval(subst(e2,eval(e1),x))
  ...
  case Lambda(x,ty,e) => Lambda(x,ty,e)

  case Apply(e1,e2)   => eval(e1) match {
    case Lambda(x,_,e) => apply(subst(e,eval(e2),x))
  }
}
```


Call-by-name

- Call-by-value may evaluate expressions unnecessarily (leading to nontermination in the worst case)

$$(\lambda x.42)\ \mathit{loop} \mapsto (\lambda x.42)\ \mathit{loop} \mapsto \cdots$$

- An alternative: substitute expressions *before* evaluating

$$(\lambda x.42)\ \mathit{loop} \mapsto 42$$

- To do this, *remove* second administrative rule, and *generalize* the computational rule

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{}{(\lambda x.\ e_1)\ e_2 \mapsto e_1[e_2/x]}$$

- This evaluation strategy is called *call-by-name* (the "name" is the expression)

Example, revisited

- Consider $(\lambda x.x \times x)\ (1 + 2 \times 3)$
- Then in call-by-name we can derive:

$$\overline{(\lambda x.x \times x)\ (1 + 2 \times 3) \mapsto (1 + (2 \times 3)) \times (1 + (2 \times 3))}$$

- The rest is standard:

$$\begin{aligned}(1 + (2 \times 3)) \times (1 + (2 \times 3)) &\mapsto (1 + 6) \times (1 + (2 \times 3)) \\ &\mapsto 7 \times (1 + (2 \times 3)) \\ &\mapsto 7 \times (1 + 6) \\ &\mapsto 7 \times 7 \mapsto 49\end{aligned}$$

- Notice that we recompute the argument twice!

Interpreting call-by-name

We substitute expressions for variables *before* evaluating.

```
  def eval (e: Expr): Value = e match {
  ...
  case Let(x,e1,e2 ) => eval(subst(e2,e1,x))
  ...
  case Lambda(x,ty,e) => Lambda(x,ty,e)

  case Apply(e1,e2) => eval(e1) match {
    case Lambda(x,_,e) => eval(subst(e,e2,x))
  }
}
```


Call-by-name in Scala

- In Scala, can flag an argument as being passed by name by writing => in front of its type
- Such arguments are evaluated only when needed (but may be evaluated many times)

```
scala> def byName(x : => Int) = x + x
byName: (x: => Int)Int
scala> byName({ println("Hi␣there!"); 42})
Hi there!
Hi there!
res1: Int = 84
```

- This can be useful; sometimes we actually want to re-evaluate an expression (see next week's tutorial)

Simulating call-by-name

- Using functions, we can simulate passing $e : \tau$ by name in a call-by-value language
- Simply pass it as a "delayed" expression $\lambda().e : \mathtt{unit} \to \tau$.
- When its value is needed, apply to ().
- Scala's "by name" argument passing is basically syntactic sugar for this (using annotations on types to decide when to silently apply to ())

Comparison

- Call-by-value evaluates every expression at most once
  - ... whether or not its value is needed
  - Performance tends to be more predictable
  - Side-effects happen predictably
- Call-by-name only evaluates an expression if its value is *needed*
  - Can be faster (or even avoid infinite loop), if not needed
  - But may evaluate multiple times if needed more than once
  - Reasoning about performance requires understanding when expressions are needed
  - Side-effects may happen multiple times or not at all!

Best of both worlds?

- A third strategy: evaluate each expression when it is needed, but then *save the result*
- If an expression's value is never needed, it never gets evaluated
- If it is needed many times, it's still only evaluated once.
- This is called *call-by-need* (or sometimes *lazy*) evaluation.

Laziness in Scala

- Scala provides a `lazy` keyword
- Variables declared `lazy` are not evaluated until needed
- When they are evaluated, the value is *memoized* (that is, we store it in case of later reuse).

```
scala> lazy val x = {println("Hello"); 42}
x: Int = <lazy>
scala> x + x
Hello
res0: Int = 84
```


Laziness in Scala

- Actually, laziness can also be *emulated* using references and variant types:

```
class Lazy[A](a: => A) {
  private var r: Either[A,() => A] = Right{() => a}
  def force = r match {
    case Left(a) => a
    case Right(f) => {
      val a = f()
      r = Left(a)
      a
    }
  }
}
```

Call-by-need

- The semantics of call-by-need is a little more complicated.
- We want to *share* expressions to avoid recomputation of needed subexpressions
- We can do this using a "memo table" $\sigma : Loc \to Expr$
  - (similar to the *store* we used for references)
- Idea: When an expression e is bound to a variable, replace it with a *label* ℓ bound to e in σ
  - The labels are *not* regarded as values, though.
  - When we try to evaluate the label, look up the expression in the store and evaluate it

Rules for call-by-need

$\boxed{\sigma, e \mapsto \sigma', e'}$

$$\frac{}{\sigma, (\lambda x.e_1)\ e_2 \mapsto \sigma[\ell := e_2], e_1[\ell/x]}$$

$$\frac{}{\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mapsto \sigma[\ell := e_1], e_2[\ell/x]}$$

$$\frac{}{\sigma[\ell := v], \ell \mapsto \sigma[\ell := v], v} \qquad \frac{\sigma, e \mapsto \sigma', e'}{\sigma[\ell := e], \ell \mapsto \sigma'[\ell := e'], \ell}$$

- When we reduce a function application or `let`, add expression to the memo table and replace with label
- When we encounter the label, look up its value or evaluate it (if not yet evaluated)

Rules for call-by-need

As with $\mathsf{L}_{\mathsf{Ref}}$, we also need to adjust all of the rules to handle σ.

$\boxed{\sigma, e \mapsto \sigma', e'}$

$$\frac{\sigma, e_1 \mapsto \sigma', e_1'}{\sigma, e_1 \oplus e_2 \mapsto \sigma', e_1' \oplus e_2} \qquad \frac{\sigma, e_2 \mapsto \sigma', e_2'}{\sigma, v_1 \oplus e_2 \mapsto \sigma', v_1 \oplus e_2'}$$

$$\frac{}{\sigma, v_1 + v_2 \mapsto \sigma, v_1 +_{\mathbb{N}} v_2} \qquad \frac{}{\sigma, v_1 \times v_2 \mapsto \sigma, v_1 \times_{\mathbb{N}} v_2}$$

$$\vdots$$

Example, revisited again

- Consider $(\lambda x.x \times x)\ (1 + 2 \times 3)$
- Then we can derive:

$$\frac{}{[], (\lambda x.x \times x)\ (1 + 2 \times 3) \mapsto [\ell = 1 + (2 \times 3)], \ell \times \ell}$$

- Next, we have:

$$[\ell = 1 + (2 \times 3)], \ell \times \ell \mapsto [\ell = 1 + 6], \ell \times \ell \mapsto [\ell = 7], \ell \times \ell$$

- Finally, we can fill in the ℓ labels:

$$[\ell = 7], \ell \times \ell \mapsto [\ell = 7], 7 \times \ell \mapsto [\ell = 7], 7 \times 7 \mapsto [\ell = 7], 49$$

- Notice that we compute the argument only once (but only when its value is needed).

Pure functional programming

- Call-by-name/call-by-need interact *badly* with side-effects
- On the other hand, they support very strong *equational* reasoning about programs
- Haskell (and some other languages) are *pure*: they adopt lazy evaluation, and forbid **any** side-effects!
- This has strengths and weaknesses:
  - (+) Easier to optimize, parallelize because side-effects are forbidden
  - (+) Can be faster
  - (-) but memoization has overhead (e.g. memory leaks) and performance is less predictable
  - (-) Dealing with I/O, exceptions etc. requires major rethink

I/O in Haskell

- Dealing with I/O and other side-effects in Haskell was a long-standing challenge
- Today's solution: use a type constructor `IO a` to "encapsulate" side-effecting computations

  ```
  do { x <- readLn::IO Int ; print x }
  123
  123
  ```

- Note: `do`-notation is also a form of *comprehension*
- Haskell's *monads* provide (equivalents of) the `map` and `flatMap` operations

Lazy data structures

- We have (so far) assumed eager evaluation for data structures (pairs, variants)
  - e.g. a pair is fully evaluated to a value, even if both components are not needed
- However, alternative (lazy) evaluation strategies can be considered for data structures too
  - e.g. could consider a pair (e_1, e_2) to be a value; we only evaluate e_1 if it is "needed" by applying `fst`:

    ```
    ghci> fst (42, undefined) == 42
    ```

- An example: *streams* (see next week's tutorial)

```
ghci> let ones = 1::ones
ghci> take 10 ones
```


Summary

- We are continuing our tour of language-design issues
- Today we covered:
  - Call-by-value (the default)
  - Call-by-name
  - Call-by-need and lazy evaluation
- Next time:
  - Exceptions
  - Control abstractions

Elements of Programming Languages

Lecture 16: Exceptions and Control Abstractions

James Cheney

University of Edinburgh

November 22, 2016

Overview

- We have been considering several high-level aspects of language design:
  - Type soundness
  - References
  - Evaluation order
- Today we complete this tour and examine:
  - Exceptions
  - Tail recursion
  - Other control abstractions

Exceptions

- In earlier lectures, we considered several approaches to *error handling*
- *Exceptions* are another popular approach (supported by Java, C++, Scala, ML, Python, etc.)
- The `throw e` statement *raises an exception* e
- A try/catch block runs a statement; if an exception is raised, control transfers to the corresponding *handler*

```
try { ... do something ... }
catch (IOException e)
  {... handle exception e ...}
catch (NullPointerException e)
  {... handle another exception...}
```


`finally` and resource cleanup

- What if the `try` block allocated some resources?
- We should make sure they get deallocated!
- `finally` clause: gets run at the end whether or not exception is thrown

```
InputStream in = null;
try { in = new FileInputStream(fname);
      ... do something with in ... }
catch (IOException exn) {...}
finally { if(in != null)
            in.close(); }
```

- Java 7: "try-with-resources" encapsulates this pattern, for resources implementing `AutoCloseable` interface

throws clauses

- In Java, potentially unhandled exceptions typically need to be *declared* in the types of methods

```
void writeFile(String filename)
    throws IOException {
  InputStream in = new FileInputStream(filename);
  ... write to file ...
  in.close();
}
```

- This means programmers using such methods know that certain exceptions need to be handled
- Failure to handle or declare an exception is a type error!
  - (however, certain *unchecked exceptions* / errors do not need to be declared, e.g. `NullPointerException`)

Exceptions in Scala

- As you might expect, Scala supports a similar mechanism:

```
try { ... do something ... }
catch {
  case exn: IOException =>
    ... handle IO exception...
  case exn: NullPointerException =>
    ... handle null pointer exception...
} finally { ... cleanup ...}
```

- Main difference: The `catch` block is just a Scala pattern match on exceptions
  - Scala allows pattern matching on *types* (via `isInstanceOf`/`asInstanceOf`)
- Also: `throws` clauses not required

Exceptions for shortcutting

- We can also use exceptions for "normal" computation

```
def product(l: List[Int]) = {
  object Zero extends Throwable
  def go(l: List[Int]): Int = l match {
    case Nil => 1
    case x::xs =>
      if (x == 0) {throw Zero} else {x * go(xs)}
  }
  try { go(l) }
  catch { case Zero => 0 }
}
```

- potentially saving a lot of effort if the list contains 0

Exceptions in practice

- Java:
  - Exceptions are subclasses of `java.lang.Throwable`
  - Method types must declare (most) possible exceptions in `throws` clause
  - compile-time error if an exception can be raised and not caught or declared
  - multiple "catch" blocks; "finally" clause to allow cleanup
- Scala:
  - doesn't require declaring thrown exceptions: this becomes especially painful in a higher-order language...
  - "catch" does pattern matching

Modeling exceptions

- We will formalize a simple model of exceptions:

$$e ::= \cdots \mid \texttt{raise}\ e \mid e_1\ \texttt{handle}\ \{x \Rightarrow e_2\}$$

- Here, $\texttt{raise}\ e$ throws an arbitrary value as an "exception"
- while $e_1\ \texttt{handle}\ \{x \Rightarrow e_2\}$ evaluates e_1 and, if an exception is thrown during evaluation, binds the value v to x and evaluates e.
- Define $\mathsf{L}_{\mathsf{Exn}}$ as $\mathsf{L}_{\mathsf{Rec}}$ extended with exceptions

Exceptions and types

- Exception constructs are straightforward to typecheck:

$$\tau ::= \cdots \mid \texttt{exn}$$

- Usually, the exn type is extensible (e.g. by subclassing)

$\boxed{\Gamma \vdash e : \tau}$ for $\mathsf{L}_{\mathsf{Exn}}$

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \texttt{exn} \vdash e_2 : \tau}{\Gamma \vdash e_1\ \texttt{handle}\ \{x \Rightarrow e_2\} : \tau}$$

- Note: $\texttt{raise}\ e$ can have any type! (because $\texttt{raise}\ e$ never returns)
- The return types of e_1 and e_2 in handler must match.

Interpreting exceptions

- We can extend our Scala interpreter for $\mathsf{L}_{\mathsf{Rec}}$ to manage exceptions as follows:

```
case class ExceptionV(v: Value) extends Throwable
def eval(e: Expr): Value = e match {
  ...
  case Raise(e: Expr) => throw (ExceptionV(eval(e)))
  case Handle(e1: Expr, x: Variable, e2:Expr) =>
    try {
      eval(e1)
    } catch (ExceptionV(v)) {
      eval(subst(e2,v,x))
    }
```

- This might seem a little circular!

Semantics of exceptions

- To formalize the semantics of exceptions, we need an auxiliary judgment e raises v
- Intuitively: this says that expression e does not finish normally but instead raises exception value v

$\boxed{e \text{ raises } v}$

$$\frac{}{\texttt{raise}\ v \text{ raises } v} \qquad \frac{e_1 \text{ raises } v}{e_1 \oplus e_2 \text{ raises } v} \qquad \frac{e_2 \text{ raises } v}{v_1 \oplus e_2 \text{ raises } v}$$

$$\frac{e \text{ raises } v}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \text{ raises } v} \qquad \cdots$$

- The most interesting rule is the first one; the rest are "administrative"

Semantics of exceptions

- We can now define the small-step semantics of `handle` using the following additional rules:

$\boxed{e \mapsto e'}$

$$\frac{e_1 \mapsto e_1'}{e_1 \; \texttt{handle} \; \{x \Rightarrow e_2\} \mapsto e_1' \; \texttt{handle} \; \{x \Rightarrow e_2\}}$$

$$\frac{}{v_1 \; \texttt{handle} \; \{x \Rightarrow e_2\} \mapsto v_1}$$

$$\frac{e_1 \text{ raises } v}{e_1 \; \texttt{handle} \; \{x \Rightarrow e_2\} \mapsto e_2[v/x]}$$

- If e_1 steps normally to e_1', take that step
- If e_1 raises an exception v, substitute it in for x and evaluate e_2

Tail recursion

- A function call is a *tail call* if it is the last action of the calling function. If every recursive call is a tail call, we say `f` is *tail recursive*.
- For example, this version of `fact` is not tail recursive:

```
def fact1(n: Int): Int =
  if (n == 0) {1} else {n * (fact1(n-1))}
```

- But this one is:

```
def fact2(n: Int) = {
  def go(n: Int, r: Int): Int =
    if (n == 0) {r} else {go(n-1,n*r)}
  go(n,1)
}
```


Tail recursion and efficiency

- Tail recursive functions can be compiled more efficiently
- because there is no more “work” to do after the recursive call
- In Scala, there is a (checked) annotation `@tailrec` to mark tail-recursive functions for optimization

```
def fact2(n: Int) = {
  @tailrec
  def go(n: Int, r: Int): Int =
    if (n == 0) {r} else {go(n-1,n*r)}
  go(n,1)
}
```


Continuations [non-examinable]

- Conditionals, while-loops, exceptions, “goto” are all form of *control abstraction*
- *Continuations* are a highly general notion of control abstraction, which can be used to implement exceptions (and much else).
- Material covered from here on is non-examinable.
  - just for fun!
  - (Depends on your definition of fun, I suppose)

Continuations

- A continuation is a function representing "the rest of the computation"
- Any function can be put in "continuation-passing form"
- for example

```
def fact3[A](n: Int, k: Int => A): A =
  if (n == 0) {k(1)}
  else {fact3(n-1, {m => k (n * m)})}
```

- This says: if n is 0, pass 1 to k
- otherwise, recursively call with parameters $n-1$ and $\lambda r.k(n \times r)$
- "when done, multiply the result by n and pass to k"

How does this work?

```
def fact3[A](n: Int, k: Int => A): A =
  if (n == 0) {k(1)} else {fact3(n-1, {r => k (n * r)})}
```

$$
\begin{array}{ll}
 & \mathit{fact}3(3, \lambda x.x) \\
\mapsto & \mathit{fact}3(2, \lambda r_1.(\lambda x.x)\ (3 \times r_1)) \\
\mapsto & \mathit{fact}3(1, \lambda r_2.(\lambda r.(\lambda x.x)\ (3 \times r))\ (2 \times r_2)) \\
\mapsto & \mathit{fact}3(0, \lambda r_3.(\lambda r_2.(\lambda r_1.(\lambda x.x)\ (3 \times r_1))\ (2 \times r_2))\ (1 \times r_3)) \\
\mapsto & (\lambda r_3.(\lambda r_2.(\lambda r_1.(\lambda x.x)\ (3 \times r_1))\ (2 \times r_2))\ (1 \times r_3))\ 1 \\
\mapsto & (\lambda r_2.(\lambda r_1.(\lambda x.x)\ (3 \times r_1))\ (2 \times r_2))\ (1 \times 1) \\
\mapsto & (\lambda r_1.(\lambda x.x)\ (3 \times r_1))\ (2 \times 1) \\
\mapsto & (\lambda x.x)\ (3 \times 2) \\
\mapsto & 6
\end{array}
$$

Interpreting $\mathsf{L}_{\mathsf{Arith}}$ using continuations

```
def eval[A](e: Expr, k: Value => A): A = e match {
  // Arithmetic
  case Num(n) => k(NumV(n))

  case Plus(e1,e2) =>
    eval(e1,{case NumV(v1) =>
      eval(e2,{case NumV(v2) => k(NumV(v1+v2))})})

  case Times(e1,e2) =>
    eval(e1,{case NumV(v1) =>
      eval(e2,{case NumV(v2) => k(NumV(v1*v2))})})
  ...
}
```


Interpreting L_{If} using continuations

```
def eval[A](e: Expr, k: Value => A): A = e match {
  ...
  // Booleans
  case Bool(n) => k(BoolV(n))

  case Eq(e1,e2) =>
    eval(e1,{v1 =>
      eval(e2,{v2 => k(BoolV(v1 == v2))})})

  case IfThenElse(e,e1,e2) =>
    eval(e,{case BoolV(v) =>
      if(v) { eval(e1,k) } else { eval(e2,k) } })
  ...
}
```

Interpreting L_{Let} using continuations

```
def eval[A](e: Expr, k: Value => A): A = e match {
  ...
  // Let-binding
  case Let(e1,x,e2) =>
    eval(e1,{v =>
      eval(subst(e2,v,x),k)})
  ...
}
```


Interpreting L_{Rec} using continuations

```
def eval[A](e: Expr, k: Value => A): A = e match {
  ...
  // Functions
  case Lambda(x,ty,e) => k(LambdaV(x,ty,e))
  case Rec(f,x,ty1,ty2,e) => k(RecV(f,x,ty1,ty2,e))

  case Apply(e1,e2) =>
    eval(e1, {v1 =>
      eval(e2, {v2 => v2 match {
        case LambdaV(x,ty,e) => eval(subst(e,v2,x), k)
        case RecV(f,x,ty1,ty2,e) =>
          eval(subst(subst(e,v2,x),v1,f),k)
      }})})
  ...
}
```


Interpreting L_{Exn} using continuations

To deal with exceptions, we add a second continuation `h` for handling exceptions. (Cases seen so far just pass `h` along.)

```
def eval[A](e: Expr, h: Value => A,
                      k: Value => A): A = e match {
  ...
  // Exceptions
  case Raise(e0) => eval(e0,h,h)

  case Handle(e1,x,e2) =>
    eval(e1,{v => eval(subst(e2,v,x),h,k)},k)
}
```

When raising an exception, we forget `k` and pass to `h`.
When handling, we install new handler using `e2`

Summary

- Today we completed our tour of
  - Type soundness
  - References and resource management
  - Evaluation order
  - Exceptions and control abstractions (today)
- which can interact with each other and other language features in subtle ways
- Next time:
  - review lecture
  - information about exam, reading

Elements of Programming Languages

Course review

James Cheney

University of Edinburgh

November 25, 2016

Overview

- We've now covered
  - Basic concepts: ASTs, evaluation, typing, names, scope
  - Common elements of any programming language
  - Programming in the large: components, abstractions
  - Language design issues
- Today:
  - Review of course, pointers to related reading
  - Information about the exam
  - Conclusions

Intro & Abstract syntax

- Concrete vs. Abstract Syntax
- Abstract syntax trees
- Abstract syntax of L_{Arith} in several languages
- Structural induction over syntax trees
- Reading: PFPL2 1.1; CPL 4.1, 5.4.1

Evaluation & Interpretation

- A simple interpreter for arithmetic expressions
- Evaluation judgment $e \Downarrow v$ and big-step evaluation rules
- Totality, uniqueness, and correctness of interpreter (via structural induction)
- Reading: PFPL2 2.1-3, 2.6, 7.1, CPL 5.4.2

Booleans, conditionals, types

- Boolean expressions, equality tests, and conditionals
- Typing judgment $\vdash e : \tau$
- Typing rules
- Type soundness and static vs. dynamic typing
- Reading: PFPL2 4.1-4.2, CPL 5.4.2, 6.1, 6.2

Variables and scope

- Variables: symbols denoting other things
- Substitution: replacing variables with expressions/values
- Scope and binding: introducing and using variables
- Free variables and α-equivalence
- Impact of variables, scope and binding on evaluation and typing (using `let`-binding to illustrate)
- Reading: PFPL2 1.2, 3.1-3.2, CPL 4.2, 7.1

Functions and recursion

- Named (non-recursive) functions
- Static vs. dynamic scope
- Anonymous functions
- Recursive functions
- The function type, $\tau_1 \to \tau_2$
- Reading: PFPL2 8, 19.1-2; CPL 4.2, 5.4.3

Data structures

- Pairs and pair types $\tau_1 \times \tau_2$, which combine two or more data structures
- Variant/choice types $\tau_1 + \tau_2$, which represent a choice between two or more data structures
- Special cases `unit`, `empty`
- Reading: PFPL2 10.1, 11.1, CPL 5.4.4

Records, variants and subtyping

- Records, generating from pairs to structures with named fields
- Named variants, generalizing from binary choices to named constructors (e.g. datatypes, case classes)
- Type abbreviations and definitions
- Subtyping (e.g. width subtyping, depth subtyping for records)
- Covariance and contravariance; subtyping for pair, choice, function types
- Reading: CPL 6.5; PFPL2 10.2, 11.2-3, 24.1-3

Polymorphism and type inference

- The idea of thinking of the same code as having many different types
- Parametric polymorphism: abstracting over a type parameter (variable)
- Modeling polymorphism using types $\forall A.\tau$
- High-level coverage of type inference, e.g. in Scala
- **[non-examinable]** Hindley-Milner and let-bound polymorphism
- Reading: PFPL2 16.1; CPL 6.3-4

Programs, modules and interfaces

- "Programs" as collections of definitions (with an entry point)
- Namespaces and packages: collecting related components together, using "dot" syntax to structure names; importing namespaces to allow local usage
- The idea of abstract data types: a type with associated operations, with hidden implementation
- Modules (e.g. Scala's objects) and interfaces (e.g. Scala's traits)
- What it means for a module to "implement" an interface
- Reading: CPL 9, PFPL2 42.1-2, 44.1

Objects and classes

- Objects and how they differ from records or modules: encapsulation of local state; self-reference
- Classes and how they differ from interfaces; abstract classes; dynamic dispatch
- Instantiating classes to obtain objects
- Inheritance of functionality between objects or classes; multiple inheritance and its problems
- Run-time type tests and coercions (isInstanceOf, asInstanceOf)
- Reading: CPL 10, 12.5, 13.1-2

Object-oriented functional programming

- Advanced OOP concepts:
  - inner classes, nested classes, anonymous classes/objects
  - Generics: Parameterized types and parametric polymorphism; interaction with subtyping; type bounds
  - Traits as mixins: implementing multiple traits providing orthogonal functionality; comparison with multiple inheritance
- Function types as interfaces
- List comprehensions and `map`, `flatMap` and `filter` functions
- Reading: Odersky and Rompf, Unifying Functional and Object-Oriented Programming with Scala, CACM, Vol. 57 No. 4, Pages 76-86, April 2014

Imperative programming

- L_{While}: a language with statements, variables, assignment, conditionals and loops
- Interpreting L_{While} using *state* or *store*
- Operational semantics of L_{While}
- **[non-examinable]** Structured vs unstructured programming
- **[non-examinable]** Other control flow constructs: goto, switch, break/continue
- Reading: CPL 4.4, 5.1-2, 8.1

Small-step semantics and type safety

- Small-step evaluation relation $e \mapsto e'$, and advantages over big-step semantics for discussing type safety
- Induction on derivations
- Type soundness: decoposition into *preservation* and *progress* lemmas
- Representative cases for L_{If}
- **[non-examinable]** Type soundness for L_{Rec}
- Reading: CPL 6.1-2, PFPL2 5.1-2, 2.4, 7.2, 6.1-2

References and resource management

- Reconciling references and mutability with a "functional" language like L_{Rec}
- Semantics and typing for references
- Potential interactions with subtyping; problem with reference / array types being covariant in e.g. Java
- **[non-examinable]** How references + polymorphism can violate type soundness
- Resources and allocation/deallocation
- Reading: PFPL2 35.1-3, CPL 5.4.5, 13.3

Evaluation strategies

- Evaluation order; varying small-step "administrative" rules to get left-to-right, right-to-left or unspecified operand evaluation order
- Evaluation strategies for function arguments (or more generally for expressions bound to variables):
  - Call-by-value / eager
  - Call-by-name
  - Call-by-need / lazy evaluation
- Interactions between evaluation strategies and side-effects
- Lazy data structures and pure functional programming (cf. Haskell)
- Reading: PFPL2 36.1, CPL 7.3, 8.4

Exceptions and continuations

- Exceptions, illustrated in Java and Scala (throw, try...catch...finally)
- Exceptions more formally: typing and small-step evaluation rules
- Tail recursion
- **[non-examinable]** Continuations
- Reading: CPL 8.2-3, PFPL2 29.1-3, PFPL2 30.1-2

Reading summary

- The following sections of CPL are recommended to provide high-level explanation and background:
  1, 4.1-2, 4.4, 5.4, 6.1-5, 7.1, 7.3, 8.1-4, 9, 10, 12.5, 13.1-3
- The following sections of PFPL2 are recommended to complement the formal content of the course:
  1, 2, 3.1-2, 4.1-2, 5.1-2, 6.1-2, 7.1-2, 8, 19.1-2, 10.1-2, 11.1-3, 16.1, 24.1-3, 35.1-3, 36.1, 42.1-2, 44.1
- (warning: chapter references for 1st edition differ!)
- In general, exam questions should be answerable using ideas introduced/explained in lectures or tutorials
- (please ask, if something mentioned in lecture slides is unclear and not explained in associated readings)

Exam Information

Exam format

- Written exam, 2 hours
- Three (multi-part) questions
- Answer Question 1 + EITHER Question 2 or 3
- Closed-book (no notes, etc.), but...
- Exam will **not** be about memorizing inference rules — any rules needed to construct derivations will be provided in a supplement
- Check University exam schedule!
  - Exam in December $\iff$ you are a visiting student AND only here for semester 1
  - Exam in April/May $\iff$ you are here for full academic year

Expectations

- Several typical kinds of questions...
- Show how to use / apply some technical content of the course (typing rules, evaluation,) — possibly in a slightly different setting than in lectures/assignments
- Define concepts; explain differences/strengths/weaknesses of differerent ideas in PL design
- Show how to extrapolate or extend concepts or technical ideas covered in lectures (possibly in ways covered in more detail in reading or tutorials but not in lectures)
- Explain and perform simple examples of inductive proofs (no more complex than those covered in lectures)

Sample exam

- A sample exam is available now on course web page
- Format: same as real exam
- Questions have not gone through same process, so:
  - There may be errors/typos (hopefully not on real exam)
  - The difficulty level may not be calibrated to the real exam (though I have tried to make it comparable)
- In particular: just because a topic is covered/not covered on the sample exam does NOT tell you it will be / will not be covered on the real exam!
- There will be a **exam review session** on Friday December 2 at 2:10pm (usual lecture time/place, 7 Bristo Square LT1)

Conclusions

What **didn't** we cover?

- Lots! (course is already dense as it is)
- Scala: implicits, richer pattern matching, concurrency, ...
- More generally:
  - language-support for concurrent programming (`synchronized`, threads, locks, etc.)
  - language support for other computational models (databases, parallel CPU, GPU, etc.)
  - Haskell-style type classes/overloading
  - Logic programming
  - Program verification / theorem proving
  - Analysis and optimisation
  - Implementation and compilation of modern languages
  - Virtual machines

Other relevant courses

- There is a lot more to Programming Languages than we can cover in just one course...
- The following UG4 courses cover more advanced topics related to programming languages:
  - **Advances in Programming Languages**
  - **Types and Semantics for Programming Languages**
  - **Secure Programming**
  - **Parallel Programming Languages and Systems**
  - **Compiler Optimisation**
  - **Formal Verification**
- Many potential supervisors for PL-related UG4, MSc, PhD projects in Informatics — ask if interested!

Other programming languages resources

- *Scottish Programming Languages Seminar*, http://www.dcs.gla.ac.uk/research/spls/
- EdLambda, Edinburgh's mostly functional programming meetup, http://www.edlambda.co.uk
- Informatics *PL Interest Group*, http://wcms.inf.ed.ac.uk/lfcs/research/groups-and-projects/pl/programming-languages-interest-group
- Major conferences: ICFP, POPL, PLDI, OOPSLA, ESOP, CC
- Major journals: ACM TOPLAS, Journal of Functional Programming

A final word

- This has been the second time of teaching this course *Elements of Programming Languages*
  - > 70 students registered last year, > 40 this year
- I hope you've enjoyed the course! I did, though there are still some things that probably need work...
- Please do provide feedback on the course (both what worked and what didn't)
  - Thanks in advance on behalf of future EPL students!