DD2552 Seminar 1: Functional languages, abstract syntax trees, variable binding, and inductive definitions

Karl Palmskog

KTH

Wednesday August 30, 2023

1. Introduction
2. Motivating Examples
3. Abstract Syntax, Variable Binding, and Inductive Definitions

About me

- PhD KTH, 2014; postdoc/researcher 2015-2021
- lecturer, KTH, 2021-
- research areas:
  - program verification and proof engineering
  - distributed systems (e.g., blockchains)
- other teaching:
  - DD2443 Parallel and Distributed Computing
  - Prosam
- email available on Canvas

This course

- seminar course on purely functional programming
- spans from lambda calculus to efficient data structures
- material is useful in formal methods, but no fully formal proofs
- focus on principles and (some) practice

Course material

- **Practical Foundations of Programming Languages**
    - Robert Harper
    - 2nd Edition, Cambridge University Press, 2016
    - http://www.cs.cmu.edu/~rwh/pfpl/abbrev.pdf
- research papers
- CakeML language and compiler https://cakeml.org

What is a functional language?

- Common LISP, Scheme, Standard ML, OCaml, Haskell, …
- Erlang? Java and C++ due to lambda expressions?
- according to Robert Harper, a functional language should
    - give meaning to programs independently of its target (hardware or software) platform
    - support both **computation by evaluation** and computation by execution
    - support **persistent** and ephemeral data structures
    - have a parallel cost model
    - have a rich type structure, supporting modular program development
- https://existentialtype.wordpress.com/2011/03/16/what-is-a-functional-language/

Focus of this course

- program meaning in terms of mathematical functions and operational semantics
- computation by “pure” evaluation
- persistent data structures
- rich type structures, including modular structures

Why purely functional?

- referential transparency (substituting equals for equals works)
- heaps are difficult to reason about
- pure functions useful as specifications of other programs
- parallelizes easily
- useful for message passing concurrency (values don't change)
- performance can (still) be good to reasonable
- conjectured to be most feasible way to build large formally verified systems
- established mathematical theories and tool support (e.g., proof assistants such as Coq and HOL4)

Example System: CompCert C compiler

- compiler for a realistic subset of C (Misra C with extensions)
- core functionality defined as collection of pure functions
- language specification and machine-checked proofs of correctness in Coq proof assistant
- generated code generally performs better than gcc with O1 optimization
- useful in development of safety critical embedded systems
- ACM System Software Award 2022
- https://compcert.org

Example Language: Standard ML

- strongly typed functional language with full module system
- eager evaluation
- developed by Milner et al. in 1980s-1990s
- rigorously defined semantics
- several compilers, including Poly/ML with thread support
- impure, but easy to use pure fragment

```
fun factorial n =
 if n = 0 then 1 else n * factorial (n - 1)
```

Example Language: OCaml

- strongly typed functional language with full module system
- eager evaluation
- developed by Huet, Leroy et al. 1980s-present
- compiler defined semantics
- thread support in version 5
- impure, but easy to use pure fragment

```
let rec fact n =
 if n =/ Int 0 then Int 1 else n */ fact (n -/ Int 1)
```

Example Language: Haskell

- strongly typed functional language with typeclasses
- lazy evaluation
- developed by Peyton Jones et al. 1990s-present
- compiler defined semantics (GHC)
- thread support
- pure

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Example Language: CakeML

- strongly typed functional language with restricted module system
- eager evaluation
- developed by Myreen et al., 2010s-present
- fully formalized semantics, embedded in HOL4 proof assistant
- bootstrapped compiler with machine-checked correctness
- impure, but easy to use pure fragment
- no thread support

```
fun fib n =
 case n of
   0 => 0
 | 1 => 1
 | n => fib (n - 1) + fib (n - 2)
```

1. Introduction

2. Motivating Examples

3. Abstract Syntax, Variable Binding, and Inductive Definitions

Course material

- PFPL chapter 1 and chapter 2
- https://lawrencecpaulson.github.io/papers/Aczel-Inductive-Defs.pdf
- https://www.cs.cmu.edu/~rwh/pfpl/supplements/ulc.pdf

Object language vs. meta language

- in this course, we define syntax and semantics of some (small) languages
- the language being defined is usually called the **object language**
- the language we are using to define object languages is called the **meta language**
- our typical meta language is “mathematical English”, but could also be some foundational formalism like ZF set theory or constructive type theory
- the “ML” in Standard ML stands for meta language

Grammars and derivations

Consider the following grammar:

$$\begin{aligned} \text{Expr} \to \text{Num} \mid \text{Expr} \ + \ \text{Expr} \mid \text{Expr} \ - \ \text{Expr} \\ \mid \text{Expr} \ * \ \text{Expr} \mid \text{Expr} \, / \, \text{Expr} \mid (\text{Expr}) \end{aligned}$$

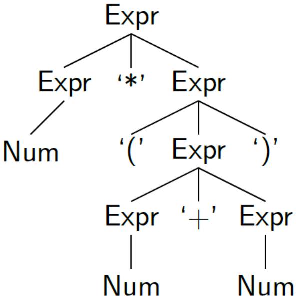
We can derive `"4*(3+5)"`:

$$\begin{aligned} \text{Expr} \to {} & \text{Expr} * \text{Expr} \to \text{Num} * \text{Expr} \to \text{Num} * (\text{Expr}) \to \\ & \text{Num} * (\text{Expr} + \text{Expr}) \to \text{Num} * (\text{Num} + \text{Expr}) \to \\ & \text{Num} * (\text{Num} + \text{Num}) \end{aligned}$$

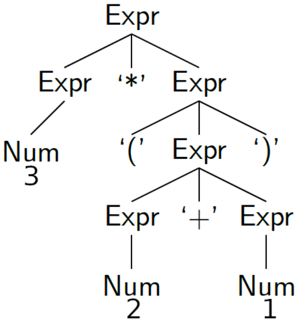
Abstract Syntax Trees

An abstract syntax tree (AST) can represent a set of derivations.

$$\begin{aligned}\underline{\text{Expr}} &\rightarrow \underline{\text{Expr}} * \text{Expr} \rightarrow \text{Num} * \underline{\text{Expr}} \rightarrow \text{Num} * (\underline{\text{Expr}}) \rightarrow \\ &\text{Num} * (\underline{\text{Expr}} + \text{Expr}) \rightarrow \text{Num} * (\text{Num} + \underline{\text{Expr}}) \rightarrow \\ &\text{Num} * (\text{Num} + \text{Num})\end{aligned}$$



Adding data to ASTs



ASTs and structural induction

Consider a more limited grammar:

$$\text{Expr} \to \text{Num} \mid \text{Expr} + \text{Expr}$$

We want to prove a property P holds for all ASTs in this grammar. It then suffices to:

- prove $P(n)$ for all numbers n
- assume $P(e)$ and $P(e')$ and prove $P(e + e')$

Why does this work? We cover all ways of forming strings according to grammar.

Variables and substitution

Consider a grammar with variables:

$$\mathrm{Expr} \to \mathrm{Num} \mid \mathrm{Expr} + \mathrm{Expr} \mid Var$$

If we have an expression e, we can *substitute* a variable x for a number n:

- $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$
- $[b/x]o(a_1, \dots, a_n) = o([b/x]a_1, \dots, [b/x]a_n)$

Variable binding is important

Let k be an unknown but fixed positive integer in the following definitions of sets:

- $L_1 = \{p^n q^n \mid n \leq k\}$
- $L_2 = \{p^n q^k \mid n > k\}$
- $L_3 = \{p^k q^n \mid k > n\}$
- $L_4 = \{p^n q^n r^n \mid n \leq k\}$
- $L_5 = \{p^n q^n r^n \mid n \geq k\}$

Variables binding in grammars

$$\text{Expr} \to \text{Num} \mid \text{Expr} + \text{Expr} \mid \text{Var} \mid \text{Let}\,\text{Var} = \text{Expr}\,\text{In}\,\text{Expr}$$

Now the second substitution rule won't work well anymore:

- $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$
- $[b/x]o(a_1, \dots, a_n) = o([b/x]a_1, \dots, [b/x]a_n)$

Solution: rename variables to avoid capture (see PFPL for details)

Inductive definitions (“judgments”)

$$\frac{\begin{array}{c} J_1 \\ \dots \\ J_n \end{array}}{J}$$

or

$$\frac{J_1 \dots J_n}{J}$$

- define relation or mathematical structure via rules
- relation name can occur in J_i (“recursive call”)
- rules can only be applied finite number of time

Inductive definition examples

$$\frac{}{n \Downarrow n} \quad \text{OS\_\_EVAL\_\_NUM}$$

$$\frac{\begin{array}{l} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 + n_2 = n \end{array}}{e_1 + e_2 \Downarrow n} \quad \text{OS\_\_EVAL\_\_PLUS}$$

Inductive derivations

- derive a judgment by reducing it to other judgments via rules
- all reductions must terminate using rules without (inductive) premises
- derivations are **trees** with the desired judgment (conclusion) as root

Ott

- tool for writing grammars and inductive rules
- exportable to LaTeX (also Coq, HOL4, Isabelle/HOL)
- https://github.com/ott-lang/ott

Ott grammar example

```
grammar
e :: e_ ::=
  | x :: :: var {{ com variable }}
  | n :: :: num {{ com number }}
  | e + e' :: :: plus {{ com plus }}
  | e * e' :: :: times {{ com times }}
  | let x := e in e' :: :: def (+ bind x in e' +)
    {{ com let }}
  | e [ e' / x ] :: M :: subst
    {{ com substitution }}
    {{ coq (subst_e [[e']] [[x]] [[e]]) }}
  | ( e ) :: S :: parentheses
    {{ coq ([[e]]) }}
```

Ott grammar using generated LaTeX

e	$::=$			
	$\|$	x		variable
	$\|$	n		number
	$\|$	$e + e'$		plus
	$\|$	$e * e'$		times
	$\|$	$\mathbf{let}\, x := e\, \mathbf{in}\, e'$	bind x in e'	let
	$\|$	$e[e'/x]$	M	substitution
	$\|$	(e)	S	

Ott rules example

```
defn
  e -> e' :: :: red :: red_
  {{ com reduction step }} by

  n1 + n2 = n
  ------------ :: plus
  n1 + n2 -> n

  e1 -> e'1
  ------------------- :: plus_l
  e1 + e2 -> e'1 + e2
```

Ott rules using generated LaTeX

$$\frac{n_1 + n_2 = n}{n_1 + n_2 \to n} \quad \text{OS\_\_RED\_\_PLUS}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \quad \text{OS\_\_RED\_\_PLUS\_\_L}$$

Semantics of expressions using rules

$$\frac{e \to e'}{n + e \to n + e'} \quad \text{OS\_\_RED\_\_PLUS\_\_R}$$

$$\frac{n_1 * n_2 = n}{n_1 * n_2 \to n} \quad \text{OS\_\_RED\_\_TIMES}$$

$$\frac{e_1 \to e_1'}{e_1 * e_2 \to e_1' * e_2} \quad \text{OS\_\_RED\_\_TIMES\_\_L}$$

$$\frac{e \to e'}{n * e \to n * e'} \quad \text{OS\_\_RED\_\_TIMES\_\_R}$$

$$\frac{e_1 \to e_1'}{\mathbf{let}\ x := e_1\ \mathbf{in}\ e_2 \to \mathbf{let}\ x := e_1'\ \mathbf{in}\ e_2} \quad \text{OS\_\_RED\_\_LET}$$

$$\frac{}{\mathbf{let}\ x := n\ \mathbf{in}\ e_2 \to e_2[n/x]} \quad \text{OS\_\_RED\_\_BIND}$$

Using the reduction relation

- we are given some expression AST e
- consider reflexive-transitive closure of $\rightarrow$ on expressions
- if $e \rightarrow^* n$, then n is the result of evaluating e
- to find and prove $e \rightarrow^* n$, we may have to do a lot of deriving

Alternative inductive relation

$$\frac{}{n \Downarrow n} \quad \text{OS\_\_EVAL\_\_NUM}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 + n_2 = n \end{array}}{e_1 + e_2 \Downarrow n} \quad \text{OS\_\_EVAL\_\_PLUS}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 * n_2 = n \end{array}}{e_1 * e_2 \Downarrow n} \quad \text{OS\_\_EVAL\_\_TIMES}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2[n_1/x] \Downarrow n_2 \end{array}}{\mathbf{let}\, x := e_1 \,\mathbf{in}\, e_2 \Downarrow n_2} \quad \text{OS\_\_EVAL\_\_LET}$$

How is this related to $\rightarrow$?