CS546
Introduction to Type Theory and Static Analysis

Lecture 4

Untyped Arithmetic

Polyvios Pratikakis

Abstract Syntax

- Abstract: a description of the AST, hides parsing details

```
t ::=
    true
    false
    if t then t else t
    0
    succ t
    pred t
    iszero t
```

- Constant terms `true`, `false`, `0` are *values*
- A language is the set of all possible terms

Language Definitions

- Inductive definition:

  The language is the set $\mathcal{T}$ of terms such that

  - {`true`, `false`, `0`} are in $\mathcal{T}$
  - if `t1` is in $\mathcal{T}$, then {`succ t1`, `pred t1`, `iszero t1`} are also in $\mathcal{T}$
  - if `t1`, `t2` and `t3` are in $\mathcal{T}$, then {`if t1 then t2 else t3`} is also in $\mathcal{T}$
  - Nothing else is in $\mathcal{T}$

Language Definitions (cont'd)

- Definition by inference rules

Axiom:
rule with no premises

$$\text{true} \in \mathcal{T} \qquad \text{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

Above the line:
premises

$$\frac{t1 \in \mathcal{T}}{\text{succ } t1 \in \mathcal{T}} \qquad \frac{t1 \in \mathcal{T}}{\text{pred } t1 \in \mathcal{T}} \qquad \frac{t1 \in \mathcal{T}}{\text{iszero } t1 \in \mathcal{T}}$$

Below the line:
conclusion

Inference rule

$$\frac{t1 \in \mathcal{T} \quad t2 \in \mathcal{T} \quad t3 \in \mathcal{T}}{\text{if } t1 \text{ then } t2 \text{ else } t3 \in \mathcal{T}}$$

Language Definitions (cont'd)

- Definition by construction

  Define set S(i)

  - $S(0) = \emptyset$
  - $S(i+1) = \{\text{true, false, 0}\}$

    $\cup \{\text{succ t1, pred t1, iszero t1} \mid \text{t1} \in S(i)\}$

    $\cup \{\text{if t1 then t2 else t3} \mid \text{t1, t2, t3} \in S(i)\}$
  - $S = \bigcup S(i)$, for all i

In OCaml

- OCaml data types are nice for AST description

```
type term =
      TmTrue
    | TmFalse
    | TmIf of term * term * term
    | TmZero
    | TmSucc of term
    | TmPred of term
    | TmIsZero of term
```

- Quite close to the abstract grammar

Defining Inductive Properties

- The set of constants in a program

Consts(`true`) = {`true`}

Consts(`false`) = {`false`}

Consts(`0`) = {`0`}

Consts(`succ t1`) = Consts(`t1`)

Consts(`pred t1`) = Consts(`t1`)

Consts(`iszero t1`) = Consts(`t1`)

Consts(`if t1 then t2 else t3`) = Consts(`t1`) ∪ Consts(`t2`) ∪ Consts(`t3`)

- Inductive definition
  - base cases for values
  - inductive cases based on *smaller* terms

In OCaml

- Data types are inductive, just pattern match!

```
let rec consts = function
      TmTrue -> [TmTrue]
    | TmFalse -> [TmFalse]
    | TmIf(t1,t2,t3) ->
        (consts t1) @ (consts t2) @ (consts t3)
    | TmZero -> [TmZero]
    | TmSucc(t1)
    | TmPred(t1)
    | TmIsZero(t1) -> consts t1
```

- Will calculate a list of all the constants in the term

Another Inductive Definition

- The size of a term

size(`true`)	= 1
size(`false`)	= 1
size(`0`)	= 1
size(`succ t1`)	= size(`t1`) + 1
size(`pred t1`)	= size(`t1`) + 1
size(`iszero t1`)	= size(`t1`) + 1
size(`if t1 then t2 else t3`)	= size(`t1`) + size(`t2`) + size(`t3`) + 1

- Counts the nodes in the AST

In OCaml

- Again, straightforward with pattern matching

```
let rec size = function
    TmTrue
  | TmFalse
  | TmZero -> 1
  | TmIf(t1,t2,t3) ->
      (size t1) + (size t2) + (size t3) + 1
  | TmSucc(t1)
  | TmPred(t1)
  | TmIsZero(t1) -> (size t1) + 1
```

- Looks familiar?

Yet Another Inductive Definition

- A term t is a numerical value

isnumerical(`true`) = `false`

isnumerical(`false`) = `false`

isnumerical(`0`) = `true`

isnumerical(`succ t1`) = isnumerical(`t1`)

isnumerical(`pred t1`) = isnumerical(t1)

isnumerical(`iszero t1`) = `false`

isnumerical(`if t1 then t2 else t3`) = `false`

- Implement in OCaml?
- The property isvalue(t) is similar

Inductive Proofs

- Given an inductive definition of terms t, prove property P(t) for all possible terms t
  - Basically, case analysis on the grammar of t
- Ordinary induction
  - Show P(t) holds for base cases
  - Assuming P(t') for n terms t1..tn, show P(t) for every inductive case constructing a term t from t1..tn
- Structural induction
  - Assuming P(t') for all immediate subterms t' of t, show P(t)
- Complete induction
  - Assuming P(t) holds for all terms t' that are smaller than t (not just immediate subterms), prove P(t)

Semantics

- Enough about syntax
- What does a program mean?
  - What does a programming language mean?
- Formal semantics of a programming language:

  A mathematical description of all possible computations of all possible programs

- Three main approaches to semantics
  - Denotational
  - Operational
  - Axiomatic

Denotational Semantics

- Define the meaning by translation to another language with known meaning
  - Equivalent to compilation
  - Defined as an interpretation function from terms to elements in a mathmatical domain (numbers, functions, etc)
  - Abstract away details of computation
- Example: [t] is the meaning of term t
  - [0] = 0
  - [succ t] = [t] + 1
  - [pred t] = [t] - 1
  - [if t1 then t2 else t3] = [t2], when [t1] is true, [t3] otherwise
  - etc.

Axiomatic Semantics

- Define the meaning of syntax using axioms
  - Invariants, properties/predicates that hold at each program point
  - Preconditions: properties that hold before execution of a term
  - Postconditions: properties that hold after evaluation of a term (if it terminates)
- Based on predicate logic
- Used to prove the correctness of programs
- Examples:
  - {true} x := 5 {!x = 5}
  - {x <> 0} z = y/x {z = y/x, x <> 0}
  - $$\frac{\{\text{P and x=5}\}\ \text{t2}\ \{\text{Q}\} \qquad \{\text{P and x<>5}\}\ \text{t3}\ \{\text{Q}\}}{\{\text{P}\}\ \text{if x=5 then t2 else t3}\ \{\text{Q}\}}$$

Operational Semantics

- Define an abstract machine that evaluates the program
  - Equivalent to an interpreter
  - Usually by term rewriting
- Machine states are just terms of the language
  - Can include other terms outside the program language e.g. terms in a language that describes memory contents
- Small-step operational semantics
  - Computation is a transition function that takes a machine state and returns the next state (executes one step of computation)
  - $t \rightarrow t'$ means term t takes a step and becomes term t'
- Big-step operational semantics
  - Computation is a transition from a machine state that includes a term, to a machine state where the term is evaluated to a resulting value
  - $t \rightarrow v$ means term t evaluates to v
  - Describes terminating executions

Operational Semantics (cont'd)

- A small-step semantics for our terms

$$\frac{}{\text{iszero } 0 \rightarrow \text{true}} \qquad \frac{\text{t1} \rightarrow \text{t1'}}{\text{iszero t1} \rightarrow \text{iszero t1'}} \qquad \frac{\text{v is a numerical value}}{\text{iszero(succ v)} \rightarrow \text{false}}$$

$$\frac{}{\text{pred } 0 \rightarrow 0} \qquad \frac{\text{t1} \rightarrow \text{t1'}}{\text{pred t1} \rightarrow \text{pred t1'}} \qquad \frac{\text{v is a numerical value}}{\text{pred(succ(v))} \rightarrow \text{v}}$$

$$\frac{}{\text{if false then t1 else t2} \rightarrow \text{t2}} \qquad \frac{\text{t1} \rightarrow \text{t1'}}{\begin{array}{c}\text{if t1 then t2 else t3} \rightarrow \\ \text{if t1' then t2 else t3}\end{array}}$$

$$\frac{}{\text{if true then t1 else t2} \rightarrow \text{t1}} \qquad \frac{\text{t1} \rightarrow \text{t1'}}{\text{succ t1} \rightarrow \text{succ t1'}}$$

In OCaml

- Each rule defines a pattern in the AST, and how to evaluate it

```
let rec step = function
    TmIsZero(TmZero) -> TmTrue
  | TmIsZero(TmSucc v) when (isnumerical v) -> TmFalse
  | TmIsZero(t1) -> let t1' = step t1 in TmIsZero(t1')
  | TmPred(TmZero) -> TmZero
  | TmPred(TmSucc(v)) when (isnumerical v) -> v
  | TmPred(t1) -> TmPred(step t1)
  | TmIf(TmTrue, t1, t2) -> t1
  | TmIf(TmFalse, t1, t2) -> t2
  | TmIf(t1, t2, t3) -> TmIf(step t1, t2, t3)
  | TmSucc(t1) -> TmSucc(step t1)
  | _ -> failwith “runtime error”
```

- That's the interpreter!

Next time

- The lambda calculus: a very simple language

  $t ::= x \mid \lambda x.t \mid t\ t$

- One kind of value, functions $\lambda x.t$ with one argument x
- One instruction, function application $t\ t$