CS 490.40
Introduction to Type Theory and Static Analysis

Lecture 4

Untyped Arithmetic
Abstract Syntax

- Abstract: a description of the AST, hides parsing details
  
  \[ t ::= \\
  \text{true} \\
  \text{false} \\
  \text{if } t \text{ then } t \text{ else } t \\
  0 \\
  \text{succ } t \\
  \text{pred } t \\
  \text{iszero } t \]

- 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

• $\{true, \text{false}, 0\}$ are in $\mathcal{T}$

• if $t_1$ is in $\mathcal{T}$, then $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\}$ are also in $\mathcal{T}$

• if $t_1$, $t_2$ and $t_3$ are in $\mathcal{T}$, then $\{\text{if } t_1 \text{ then } t_2 \text{ else } t_3\}$ is also in $\mathcal{T}$

• Nothing else is in $\mathcal{T}$
Language Definitions (cont'd)

- Definition by inference rules

\[
\begin{align*}
\text{true} & \in \mathcal{T} \\
\text{false} & \in \mathcal{T} \\
0 & \in \mathcal{T} \\
\text{succ } t1 & \in \mathcal{T} \\
\text{pred } t1 & \in \mathcal{T} \\
\text{iszero } t1 & \in \mathcal{T} \\
\text{if } t1 \text{ then } t2 \text{ else } t3 & \in \mathcal{T}
\end{align*}
\]

Axiom: rule with no premises

Above the line: premises

Below the line: conclusion

Inference rule
Language Definitions (cont'd)

• Definition by construction
Define set $S(i)$
  • $S(0) = \emptyset$
  • $S(i+1) = \{\text{true, false, 0}\}$
    $\cup \{\text{succ } t_1, \text{ pred } t_1, \text{ iszero } t_1 \mid t_1 \in S(i)\}$
    $\cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S(i)\}$
  • $S = \bigcup S(i)$, for all $i$
In OCaml

- OCaml data types are nice for AST description

```ocaml
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
  \[\text{Consts}(\text{true}) = \{\text{true}\}\]
  \[\text{Consts}(\text{false}) = \{\text{false}\}\]
  \[\text{Consts}(0) = \{0\}\]
  \[\text{Consts}(\text{succ } t1) = \text{Consts}(t1)\]
  \[\text{Consts}(\text{pred } t1) = \text{Consts}(t1)\]
  \[\text{Consts}(\text{iszero } t1) = \text{Consts}(t1)\]
  \[\text{Consts}(\text{if } t1 \text{ then } t2 \text{ else } t3) = \text{Consts}(t1) \cup \text{Consts}(t2) \cup \text{Consts}(t3)\]

- Inductive definition
  - base cases for values
  - inductive cases based on \textit{smaller} terms
In OCaml

- Data types are inductive, just pattern match!

```ocaml
def const function
    TmTrue -> [TmTrue]
    | TmFalse -> [TmFalse]
    | TmIf(t1,t2,t3) ->
      (const t1) @ (const t2) @ (const t3)
    | TmZero -> [TmZero]
    | TmSucc(t1)
    | TmPred(t1)
    | TmIsZero(t1) -> const t1
```

- Will calculate a list of all the constants in the term
Another Inductive Definition

• The size of a term

\[
\begin{align*}
\text{size(true)} &= 1 \\
\text{size(false)} &= 1 \\
\text{size(0)} &= 1 \\
\text{size(succ t1)} &= \text{size(t1)} + 1 \\
\text{size(pred t1)} &= \text{size(t1)} + 1 \\
\text{size(iszero t1)} &= \text{size(t1)} + 1 \\
\text{size(if t1 then t2 else t3)} &= \text{size(t1)} + \text{size(t2)} + \text{size(t3)} + 1
\end{align*}
\]

• 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

\[
\begin{align*}
is\text{numerical}(\text{true}) &= \text{false} \\
is\text{numerical}(\text{false}) &= \text{false} \\
is\text{numerical}(0) &= \text{true} \\
is\text{numerical}(\text{succ } t_1) &= is\text{numerical}(t_1) \\
is\text{numerical}(\text{pred } t_1) &= is\text{numerical}(t_1) \\
is\text{numerical}(\text{iszero } t_1) &= \text{false} \\
is\text{numerical}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{false}
\end{align*}
\]

• 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 $t_1..t_n$, show $P(t)$ for every inductive case constructing a term $t$ from $t_1..t_n$
- 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 mathematical domain (numbers, functions, etc)
  - Abstract away details of computation
- Example: \([t]\) is the meaning of term \(t\)
  - \([0] = 0\)
  - \([\text{succ } t] = [t] + 1\)
  - \([\text{pred } t] = [t] - 1\)
  - \([\text{if } t1 \text{ then } t2 \text{ else } t3] = [t2], \text{ when } [t1] \text{ is true, } [t3] \text{ otherwise}\)
  - 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\}
    - \{P and x=5\} t2 \{Q\}
    - \{P and x<>5\} t3 \{Q\}
    - \{P\} if x=5 then t2 else t3 \{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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iszero 0 → true</code></td>
<td><code>iszero t1 → iszero t1'</code></td>
</tr>
<tr>
<td><code>pred 0 → 0</code></td>
<td><code>pred t1 → pred t1'</code></td>
</tr>
<tr>
<td><code>if false then t1 else t2 → t2</code></td>
<td><code>if true then t1 else t2 → t1</code></td>
</tr>
<tr>
<td><code>if t1 then t2 else t3 → if t1' then t2 else t3</code></td>
<td><code>t1 → t1'</code></td>
</tr>
</tbody>
</table>

- `v` is a numerical value
  - `iszero(succ v) → false` |
  - `pred(succ(v)) → v`
In OCaml

- Each rule defines a pattern in the AST, and how to evaluate it
  
  ```ocaml
  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 \( t \ t \)