Recitation 1: Judgments, Inference, and Induction

15-312: Foundations of Programming Languages

Jeanne Luning Prak, Charles Yuan

January 17, 2018

1 Abstract Syntax Trees

The **abstract syntax tree**, or **ast**, is the central object of study in programming language theory.

Each ast is either a **variable**, which is a placeholder for an unknown object, or an **operator**, which combines other ast's. We may substitute other ast's for variables, so variables are given meaning by substitution.

A **sort** is a group of ast's corresponding to a form of syntax. We usually define sorts by listing the operators that build ast's, and variables are then allowed to stand for all the possible ast's in the sort.

Each operator has an **arity**, which describes its sort, and how many and which sorts of arguments it operates on.

Example: Consider the following sort Nat of natural numbers:

Operator	*Arity*
`zero`	()Nat
`succ`	(Nat)Nat

The operator `zero` represents the number zero, and the operator `succ`(a) represents the successor of the variable a, which is also a number.

Arities serve the same purpose for operators as types do for functions, which is to constrain the set of valid objects in the system, though they are essentially only syntactic in nature.

Now, consider the sort Exp of expressions built on top of numbers:

Operator	*Arity*
`num`	(Nat)Exp
`plus`	(Exp, Exp)Exp

The operator `num`(a) represents the number a, wrapped as an expression, and the operator `plus`($a; b$) represents the sum of expressions a and b, which is also an expression.

We can write ast's using either the **abstract** or the **concrete** syntax. The abstract method makes certain aspects of the syntax more explicit, but the concrete is easier to read and write. In this case, since the language is simple, the abstract and concrete syntax will be almost the same. We present the simplified syntax table (in "approximate" BNF representation).

Sort			*Abstract*	*Concrete*
Nat	n	$::=$	$\texttt{zero}()$	$\texttt{zero}$
			$\texttt{succ}(n)$	$\texttt{succ}(n)$
Exp	e	$::=$	$\texttt{num}(n)$	$\texttt{num}(n)$
			$\texttt{plus}(e_1; e_2)$	$\texttt{plus}(e_1; e_2)$

Examples of abstract syntax trees:

$$\begin{aligned}
\texttt{zero} &\text{ is of sort } \mathsf{Nat}\text{, representing zero}\\
\texttt{succ}(\texttt{succ}(\texttt{zero})) &\text{ is of sort } \mathsf{Nat}\text{, representing 2}\\
\texttt{num}(\texttt{succ}(\texttt{zero})) &\text{ is of sort } \mathsf{Exp}\text{, representing 1}\\
\texttt{plus}(\texttt{num}(\texttt{zero}); \texttt{num}(\texttt{succ}(\texttt{zero}))) &\text{ is of sort } \mathsf{Exp}\text{, representing } 0+1
\end{aligned}$$

2 Judgments

A **judgment** is an assertion about a property of an ast or a relationship between ast's. We write a judgment J about an ast a as $a\ J$ or $J\ a$. Judgments may also relate multiple entities.

Examples of judgments:

$n\ \mathsf{nat}$	n is a natural number
$e : \tau$	expression e has type τ
$e \Downarrow v$	expression e evaluates to value v
e is e'	expression e is identical to e'

3 Inference Rules

An **inference rule** consists of a set of judgments above the line, which are known as **premises**, and a single judgment below the line, known as the **conclusion**:

$$\frac{a\ J_1 \quad \ldots \quad a\ J_n}{a\ J}$$

A rule that does not have any premises is an **axiom**:

$$\frac{}{a\ J}$$

An **inductive definition** is a set of inference rules that completely describes a judgment over the possible ast's.

Examples of inductive definitions:

Definition of natural numbers:

$$\frac{}{\texttt{zero}\ \mathsf{nat}}\ (\mathsf{n_z}) \qquad \frac{a\ \mathsf{nat}}{\texttt{succ}(a)\ \mathsf{nat}}\ (\mathsf{n_s})$$

These notes are derived from previous course notes and Chapter 2 of *Practical Foundations for Programming Languages.*

Definition of odd and even:

$$\frac{}{\mathsf{zero\ even}}\ (\mathsf{e_z}) \qquad \frac{a\ \mathsf{even}}{\mathsf{succ}(a)\ \mathsf{odd}}\ (\mathsf{o_s}) \qquad \frac{a\ \mathsf{odd}}{\mathsf{succ}(a)\ \mathsf{even}}\ (\mathsf{e_s})$$

What is the difference here between Nat and nat? Both pertain to the natural numbers, but Nat is a syntactic collection whereas a nat is a logical statement about a. We merely selected the most obvious rules for the definition of nat, but the two concepts are otherwise unrelated.

4 Derivations

A **derivation** begins with a (possibly empty) sequence of premises and applies inference rules until it reaches a conclusion. A derivation is a constructive method of proof, and the result of one derivation can be used in another.

Example: 3 is a natural number. Proof:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\mathsf{zero\ nat}}\ (\mathsf{n_z})}{\mathsf{succ(zero)\ nat}}\ (\mathsf{n_s})}{\mathsf{succ(succ(zero))\ nat}}\ (\mathsf{n_s})}{\mathsf{succ(succ(succ(zero)))\ nat}}\ (\mathsf{n_s})$$

5 Rule Induction

A **property** $\mathcal{P}(a)$ is an arbitrary statement about an ast a.

Suppose we wish to show that if the judgment $a\ J$ is derivable, then the property $\mathcal{P}(a)$ holds. We may use a method of proof known as **rule induction**, which is similar to inductive proofs by case analysis you have previously seen.

To prove that $\mathcal{P}$ holds when J is derivable, it is enough to prove that $\mathcal{P}$ respects (is closed under) the rules defining the judgment J. More precisely, the principle of rule induction is:

To show that $\mathcal{P}$ holds over all ast's for which J holds, it is enough to show that:

For each rule
$$\frac{a_1\ J_1 \quad \dots \quad a_k\ J_k}{a\ J}$$

If $a_1\ J_1 \dots a_k\ J_k$ and $\mathcal{P}(a_1) \dots \mathcal{P}(a_k)$ hold, then $\mathcal{P}(a)$ holds.

We need only repeat for each relevant rule to complete the proof.

Example: Prove the following:

If $\mathsf{succ}(a)$ nat, then a nat.

To prove this, it suffices to prove the following property:

$\mathcal{P}(a)$: if a nat and $a = \mathsf{succ}(b)$, then b nat.

For a proof by rule induction on our above definition of nat, we need to prove the following:

1. $\mathcal{P}(\mathsf{zero})$ $(\mathsf{n_z})$
2. For every a, if a nat and $\mathcal{P}(a)$ then $\mathcal{P}(\mathsf{succ}(a))$ $(\mathsf{n_s})$

Now we prove them:

1. WTS: $\mathcal{P}(\texttt{zero})$. $\texttt{zero}$ is not of the form $\texttt{succ}(b)$. Thus, $\mathcal{P}(\texttt{zero})$ holds vacuously.
2. WTS: $\mathcal{P}(\texttt{succ}(a))$.

a nat	[by Inductive Hypothesis]
$\texttt{succ}(a) = \texttt{succ}(b)$ for some b	[Take $b = a$]
b nat	[Since a nat]
Thus, we have $\mathcal{P}(\texttt{succ}(a))$.	

■

6 Simultaneous Induction

Often, however, the simple rule induction described above is not enough to complete our proof. Inductive definitions often have premises with judgements that are different from the judgment in the conclusion. An example of this is even and odd numbers. The judgment odd relies on even and vice versa.

$$\frac{}{\texttt{zero}\ \mathsf{even}}\ (\mathsf{e_z}) \qquad \frac{a\ \mathsf{even}}{\texttt{succ}(a)\ \mathsf{odd}}\ (\mathsf{o_s}) \qquad \frac{b\ \mathsf{odd}}{\texttt{succ}(b)\ \mathsf{even}}\ (\mathsf{e_s})$$

To prove properties of such ast's, we use a process called **simultaneous induction**. We simultaneously prove two properties, $\mathcal{P}$ and $\mathcal{Q}$, one for each judgement. For even and odd numbers we can write the principle of induction, which looks like:

> To prove two properties $\mathcal{P}$ and $\mathcal{Q}$ for even and odd numbers respectively, we need to prove the following, one for each rule:
>
> 1. $\mathcal{P}(\texttt{zero})$ $(\mathsf{e_z})$
> 2. If a even and $\mathcal{P}(a)$, then $\mathcal{Q}(\texttt{succ}(a))$ $(\mathsf{o_s})$
> 3. If b odd and $\mathcal{Q}(b)$, then $\mathcal{P}(\texttt{succ}(b))$ $(\mathsf{e_s})$

If you set out initially only to prove some property $\mathcal{P}$, you may need to *strengthen* the proof by also coming up with some appropriate $\mathcal{Q}$ to prove.

Example: Prove the following:

> $\mathcal{P}(a)$: If a even, either a is $\texttt{zero}$ or $a = \texttt{succ}(b)$ where b odd.

We cannot prove directly based on even, so we need another property:

> $\mathcal{Q}(b)$: If b odd, $b = \texttt{succ}(a)$ where a even.

Now we proceed by simultaneous induction:

1. WTS: $\mathcal{P}(\texttt{zero})$.

$\texttt{zero}$ is $\texttt{zero}$.	
$\texttt{zero}$ even.	[by $(\mathsf{e_z})$]
Thus, we have $\mathcal{P}(\texttt{zero})$.	

2. Assume a even and $\mathcal{P}(a)$. WTS: $\mathcal{Q}(\mathtt{succ}(a))$

   a even [by I.H.]
   $\mathtt{succ}(a)$ odd [by ($\mathsf{o_s}$)]
   Thus, we have $\mathcal{Q}(\mathtt{succ}(a))$.

3. Assume b odd and $\mathcal{Q}(b)$. WTS: $\mathcal{P}(\mathtt{succ}(b))$

   b odd [by I.H.]
   $\mathtt{succ}(b)$ even [by ($\mathsf{e_s}$)]
   Thus, we have $\mathcal{P}(\mathtt{succ}(b))$.

■

7 Structural Induction

Rule induction is the use of proof over individual *rules* to show that a property holds for all ast's satisfying an entire inductively defined *judgment*. There is an analogous proof technique, **structural induction**, that uses proof over individual *operators* to show that a property holds for all ast's in an entire *sort*.

The principle of induction is as follows. It is simplified from the presentation in the textbook in Chapter 1.1:

> To show that $\mathcal{P}$ holds over all ast's in the sort S, it is enough to show that:
>
> 1. If x is a variable of sort S, then $\mathcal{P}(x)$.
> 2. For every operator o in S, if, for every argument a_i in the arity of o, $\mathcal{P}(a_i)$ holds, then $\mathcal{P}(o(a_1; \ldots; a_n))$ holds.

Structural induction is a fitting name for this technique, as it decomposes the abstract syntax tree structurally into variables and operators.

How do you decide whether to use rule or structural induction? It usually depends on the claim you wish to prove. For example, consider the claim "if $\mathtt{succ}(a)$ nat, then a nat" from before. We have seen how to prove this using rule induction. Does it make sense to try to prove it using structural induction? Why or why not?