Recitation 1:
Inference, Induction, and Lambda Calculus
15-312: Foundations of Programming Languages
Jeanne Luning Prak, Charles Yuan, Jeffrey Chen
Updated August 27, 2019

1 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 $aJ$ or $J a$. Judgments may also relate multiple entities.

Examples of judgments:

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

2 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:

$$
\frac{}{z \text{ nat} (n_z)} \quad \frac{a \text{ nat} (n_a)}{s(a) \text{ nat} (n_a)}
$$

Definition of odd and even:

These notes are derived from previous course notes and Chapter 2 of _Practical Foundations for Programming Languages_.

1
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.

3 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:

\[ \frac{n_z}{\text{z nat}} \]
\[ \frac{n_s}{\text{s(z) nat}} \]
\[ \frac{n_s}{\text{s(s(z)) nat}} \]

4 Rule Induction

A property \( P(a) \) is an arbitrary statement about an ast \( a \).

Suppose we wish to show that if the judgment \( a \ J \) is derivable, then the property \( P(a) \) 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 \( P \) holds when \( J \) is derivable, it is enough to prove that \( P \) respects (is closed under) the rules defining the judgment \( J \). More precisely, the principle of rule induction is:

To show that \( P \) holds over all ast's for which \( J \) holds, it is enough to show that:

For each rule

\[ \frac{a_1 \ J_1 \ldots \ a_k \ J_k}{a \ J} \]

If \( a_1 \ J_1 \ldots \ a_k \ J_k \) and \( P(a_1) \ldots P(a_k) \) hold, then \( P(a) \) holds.

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

Example: Define the sum judgment by

\[ \frac{\text{sum}(z, n, n)}{(\text{sum}_z)} \]
\[ \frac{\text{sum}(m, n, p)}{(\text{sum}_s, m, n, p)} \]

Prove that \( \text{sum} \) is unique: if \( \text{sum}(m, n, p) \) and \( \text{sum}(m, n, p') \), then \( p = p' \).

To prove this using rule induction, we first need to decide on which judgments to induct on. One approach is to do a nested rule induction on the definition of \( \text{sum}(m, n, p) \). The high level argument is as follows: given \( \text{sum}(m, n, p) \) and \( \text{sum}(m, n, p') \), first induct on the derivation of \( \text{sum}(m, n, p) \), then induct on the derivation of \( \text{sum}(m, n, p') \), and show that these two derivations must both arrive at the same value \( p = p' \).
1. Case $\text{sum}(z, n, n)$:

WTS: If $\text{sum}(z, n, p)$, then $p = n$.

Proof by rule induction on the definition of $\text{sum}(m, n, p)$:

a) Case $\text{sum}(z, n, n)$: Then $p = n$ by the conclusion of the rule.

b) Case $\text{sum}(\text{s}(m), n, \text{s}(p))$: $\text{sum}(z, n, p)$ is not of the form $\text{sum}(\text{s}(m), n, \text{s}(p))$, so this case is vacuous.

2. Case $\text{sum}(\text{s}(m), n, \text{s}(p))\text{sum}(m, n, p)$:

IH: If $\text{sum}(m, n, p')$, then $p = p'$.

WTS: If $\text{sum}(\text{s}(m), n, p'')$, then $\text{a}(p) = p''$.

Proof by rule induction on the definition of $\text{sum}(m, n, p)$:

a) Case $\text{sum}(z, n, n)$: $\text{sum}(\text{s}(m), n, p'')$ is not of the form $\text{sum}(z, n, n)$, so this case is vacuous.

b) Case $\text{sum}(\text{s}(m), n, \text{s}(p'))$:

\[
\begin{align*}
\text{sum}(m, n, p') & \quad \text{By premise of rule} \\
p & = p' \quad \text{By Inductive Hypothesis} \\
\text{s}(p) & = \text{s}(p') \quad \text{Since } p = p' \\
\text{s}(p) & = p'' \quad \text{Since } p'' \text{ is } \text{s}(p')
\end{align*}
\]

5 The “Untyped” Lambda Calculus

The Untyped Lambda Calculus, also called $\Lambda$, only has three possible expressions:

\[
x \quad \text{variable} \\
\lambda(x.e) \quad \text{abstraction} \\
e_1(e_2) \quad \text{application}
\]

We will often use the simpler notation $\lambda x.e$ to represent a lambda term, in accordance with most literature.

Despite its simplicity, $\Lambda$ is remarkably expressive. It is a Turing-complete language, capable of expressing any computation that a Turing machine, or any other commonly accepted model of computation, can. This is due to the fact that any expression in any other language can be encoded in $\Lambda$. Additionally, it is possible to define general recursion in $\Lambda$ through the use of fixed-point combinators, the most famous of which is the $Y$ combinator.

5.1 Substitution

The definition of evaluation in the lambda calculus, $\beta$-reduction, requires usage of substitution.
As discussed during lecture, there is a pitfall in defining substitution in a naive fashion. Consider substitution on this ML expression:

\[ \frac{x}{y} \text{fn } x \Rightarrow y \mapsto \text{fn } x \Rightarrow x \]

Here, \( y \) was free and we attempt to substitute, but directly substituting has allowed us to turn a constant function into the identity function, which is absurd. This situation is known as \textbf{capture} of the variable \( y \) by the binding lambda. We must use a more restrictive rule when it comes to lambda abstractions:

\[ \frac{e'}{x} \lambda(y.e) \text{ substitutes } e' \text{ for } x \text{ in } e \text{ only if } x \neq y \text{ and } y \text{ is not free in } e' \]

The requirement that \( y \) not be free in \( e' \) is known as \textbf{freshness} of \( y \). We may solve the freshness requirement in two ways: one is through \textbf{\( \alpha \)-conversion}, and another is the use of \textbf{de Bruijn indices}, which you will see on Assignment 1.