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 $a \vdash J$ or $J a$. Judgments may also relate multiple entities.

Examples of judgments:

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

$$
\begin{array}{c}
\vdash a \, J_1 \\
\vdots \\
\vdash a \, J_n \\
\hline \\
\vdash a \, J
\end{array}
$$

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

$$
\begin{array}{c}
\vdash a \, J
\end{array}
$$

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:

$$
\begin{align*}
\text{z nat} & \quad (n_z) \\
\text{s(a) nat} & \quad (n_s)
\end{align*}
$$

Definition of odd and even:

These notes are derived from previous course notes and Chapter 2 of Practical Foundations for Programming Languages.
What is the difference here between \texttt{Nat} and \texttt{nat}? Both pertain to the natural numbers, but \texttt{Nat} is a syntactic collection whereas \texttt{a nat} is a logical statement about \texttt{a}. We merely selected the most obvious rules for the definition of \texttt{nat}, but the two concepts are otherwise unrelated.

3 Derivations

A \texttt{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.

\textbf{Example:} 3 is a natural number. Proof:

\[
\frac{z \ \text{even}}{z \ \text{nat}} \\
\frac{s(z) \ \text{odd}}{s(z) \ \text{nat}} \\
\frac{s(s(z)) \ \text{even}}{s(s(z)) \ \text{nat}}
\]

4 Rule Induction

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

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

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

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

\[
\text{For each rule } a_1 J_1 \ldots a_k J_k \Rightarrow a J
\]

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

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

\textbf{Example:} Define the sum judgment by

\[
\begin{align*}
\text{sum}(z,n,n) & \quad \text{sum}(m,n,p) \\
\text{sum}(s(m),n,s(p)) & \quad \text{sum}(s(a(z)),n,a(s(z)))
\end{align*}
\]

Prove that \texttt{sum} is unique: if \texttt{sum}(m,n,p) and \texttt{sum}(m,n,p'), then \texttt{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 \texttt{nested} rule induction on the definition of \texttt{sum}(m,n,p). The high level argument is as follows: given \texttt{sum}(m,n,p) and \texttt{sum}(m,n,p'), first induct on the derivation of \texttt{sum}(m,n,p), then induct on the derivation of \texttt{sum}(m,n,p'), and show that these two derivations must both arrive at the same value \texttt{p = p}'.

\[
\frac{z \ \text{even}}{z \ \text{nat}} \\
\frac{s(a) \ \text{odd}}{s(a) \ \text{nat}} \\
\frac{s(a) \ \text{even}}{s(a) \ \text{nat}}
\]
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 $\text{sum}(s(m), n, s(p))$: $\text{sum}(z, n, p)$ is not of the form $\text{sum}(s(m), n, s(p))$, so this case is vacuous.

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

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

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

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

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

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

$$\text{sum}(m, n, p') \quad \text{By premise of rule}$$

$$p = p' \quad \text{By Inductive Hypothesis}$$

$$s(p) = s(p') \quad \text{Since } p = p'$$

$$s(p) = p'' \quad \text{Since } p'' \text{ is } s(p')$$

$\blacksquare$

5 The “Untyped” Lambda Calculus

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

- \( x \) variable
- \( \lambda(x.e) \) abstraction
- \( e_1(e_2) \) 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.