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 $Ja$. 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 $v$
- $e$ 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:

$$aJ_1 \ldots aJ_n \over aJ$$

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

$$aJ$$

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:

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

Definition of odd and even:
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:

\[
\begin{aligned}
&\frac{z \text{ even}}{(e_z)} \\
&\frac{a \text{ even}}{(o_s)} \\
&\frac{a \text{ odd}}{(e_s)} \\
&\frac{s(a) \text{ odd}}{(o_s)} \\
&\frac{s(a) \text{ even}}{(e_s)} \\
&\frac{s(z) \text{ nat}}{(n_s)} \\
&\frac{s(s(z)) \text{ nat}}{(n_s)} \\
&\frac{s(s(s(z))) \text{ nat}}{(n_s)}
\end{aligned}
\]

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: Prove the following:

If \( s(a) \text{ nat} \), then \( a \text{ nat} \).

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

\( P(a) \): if \( a \text{ nat} \) and \( a = s(b) \), then \( b \text{ nat} \).

For a proof by rule induction on our above definition of \( \text{nat} \), we need to prove the following:

1. \( P(z) \) \( (n_z) \)
2. For every \( a \), if \( a \text{ nat} \) and \( P(a) \) then \( P(s(a)) \) \( (n_s) \)

Now we prove them:

1. WTS: \( P(z) \). \( z \) is not of the form \( s(b) \). Thus, \( P(z) \) holds vacuously.
2. WTS: \( \mathcal{P}(s(a)) \).

\[
\begin{align*}
  a & \text{nat} & \text{[by Inductive Hypothesis]} \\
  s(a) &= s(b) \text{ for some } b & \text{[Take } b = a \text{]} \\
  b & \text{nat} & \text{[Since } a \text{ nat]} \\
\end{align*}
\]

Thus, we have \( \mathcal{P}(s(a)) \).

\[\blacksquare\]

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.

However, there is a pitfall in defining substitution in a naive fashion. Consider substitution on this ML expression:

\[
[x/y] \text{fn } x \Rightarrow y \leftrightarrow \text{fn } x \Rightarrow x
\]

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 capture of the variable \( y \) by the binding lambda. We must use a more restrictive rule when it comes to lambda abstractions:

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

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

5.2 \( \alpha \)-Equivalence

\( \alpha \)-equivalence is an equivalence relation on terms that allows free exchange of the choice of bound variable inside a binder. Namely:

\[
\lambda(x.e) \equiv_\alpha \lambda(y.([y/x]e))
\]
Using $\alpha$-equivalence, we may convert an expression in which some variable is not fresh to an equivalent expression in which it is fresh, and proceed with substitution.

In this course we will often speak of expressions that are $\alpha$-equivalent as equal. Such conversions will often be done implicitly.

### 5.3 Encodings in $\lambda$-Calculus

Because of the lack of datatypes in $\lambda$-calculus, it is not immediately obvious how to perform useful computation in this language. However, we may define familiar types such as Booleans, pairs, and natural numbers in the calculus. This process is called Church encoding, and is a very powerful general technique that we will many times in the course.

#### 5.3.1 Booleans

Consider the type of Booleans: it contains only two values, `true` and `false`. We need a $\lambda$-term corresponding to each of these values. It would be useful for them to capture the behavior of Booleans—that the two values express opposing truth values, and that a construct, usually denoted `if`, may use a Boolean value to distinguish two cases. There are many ways of satisfying this requirement, but here is a simple encoding:

\[
\begin{align*}
true & \triangleq \lambda x. \lambda y. x \\
false & \triangleq \lambda x. \lambda y. y \\
if(M, M_0, M_1) & \triangleq M M_0 M_1
\end{align*}
\]

This encoding is pretty clever! It defines `true` and `false` as functions that take two curried arguments and return one of the supplied arguments. `true` returns the first argument, and `false` returns the second. This means that the conditional expression is very easy to build. A Boolean expression $M$ needs only be applied to the two branches $M_0$ and $M_1$, and will itself return either the “true branch” or the “false branch” appropriately.

The Boolean type can be interpreted in two directions. The first is in terms of its constructors, which are the syntactic elements that bring a Boolean expression into existence. These constructors are `true` and `false`. The second interpretation is in terms of its destructors, which allow a Boolean expression to be used in a computation, consuming the expression in the process. Here, the destructor is `if`. A program, then, is a structure that constructs and destroys data in harmony. This theme is central to the study of programming languages, and will reveal itself in terms of the concept of types in next week’s lectures.

(We will shortly begin referring to constructors as introduction forms and destructors as elimination forms, and the interpretation with constructors as positive and the interpretation with destructors as negative. These terms come from logic.)

#### 5.3.2 Pairs

Now that we have Booleans, it is not hard to generalize to the type of pairs of values. Just as Booleans are characterized by their support of `if`, pairs are characterized by two abilities (destructors): `fst M`, returning the first element of a pair, and `snd M`, returning the second element. These two element-extractors are called projections. Rather than having the two constructors `true` and `false`, there is now only the formation constructor that takes two elements $M$ and $N$ and returns the pair $(M, N)$. 
We need to define these three constructs. Take this opportunity to try to come up with a solution!

Here is one possible definition:

\[ \langle M, N \rangle \triangleq \lambda x. x \ M \ N \]

This construction looks quite different from our last one. We define the pair of \( M \) and \( N \) as a function which applies its argument to \( M \) and \( N \) in turn. Those of you familiar with object oriented “design patterns” may recognize this as the “visitor pattern”, which supplies the data elements \( M \) and \( N \) to a “visitor” function. Each visitor function is called with the two data elements in sequence. To extract the first element from a pair, we supply \( M \) with a visitor that grabs the first argument and returns it. For the second element, we need a visitor that grabs the second argument. Does that sound familiar?

\[
\text{fst} \ M \triangleq M \ \text{true} \\
\text{snd} \ M \triangleq M \ \text{false}
\]

In fact, we may use the Booleans we just defined to act as the visitors. Using this definition, you should check that \( \text{fst} \ \langle M, N \rangle \) is equivalent to \( M \) and \( \text{snd} \ \langle M, N \rangle \) is equivalent to \( N \).

### 5.3.3 Natural Numbers

Before we attempt to define natural numbers, you should know that there are multiple popular definitions. The one that we use in class and on the assignment is due to Barendregt. It has a nice property that it is easy to define the predecessor of a number. We now explore a different, but equivalent encoding known as **Church numerals**.

The constructors for natural numbers are zero and succ\((M)\), which is the successor operation. The destructor is case\((M, M_0, M_1)\), which branches based on its argument \( M \), returning \( M_0 \) if it is zero, or \( M_1(M') \) if it is succ\((M')\).

To encode a number, we use a trick where we interpret it as the number of times we want to execute some arbitrary function \( f \) starting at some value \( b \). So, zero is encoded as \( f^0(b) \), which is the identity function applied to \( b \), which is just \( b \). One is encoded as \( f(b) \), two is encoded as \( f(f(b)) \) or \( f^2(b) \), and so on. In the encoding, we will not know what the function \( f \) or the value \( b \) are, so we accept them as arguments.

Here is the encoding:

\[
\text{zero} \triangleq \lambda b. \lambda f. b \\
\text{succ}(M) \triangleq \lambda b. \lambda f. f(M \ b \ f)
\]

In the zero case, we simply do not use \( f \) at all, and return \( b \) directly. In the successor case, we are already given a numeral \( M \) representing the ability to iterate a function \( M \) times. So, we simply pass \( M \) the arguments \( b \) and \( f \), and evaluate \( f \) one more time on the output.

Using this system, it is possible to define addition, multiplication, and a system of arithmetic.

How do we define case? For now, this is left as an exercise for you. It turns out that with Church numerals, the ability to take a predecessor (subtract) from a numeral is not completely obvious. The Barendregt numerals we studied in class make this operation easier, at the cost of making addition somewhat more complex. We will revisit the case construct in a few weeks and shine more light on this problem.
5.4 Next Lecture: The Y Combinator

So far, all the computation we have done is rather limited, with only the ability to do rudimentary arithmetic. However, $\lambda$-calculus is, in fact, capable of expressing much more powerful computations. Church’s Law asserts that it is as powerful as any other system of computation, including Turing machines. To see the idea behind expressing general recursion in $\Lambda$, we’ll use the example of a simple function that loops forever. In Standard ML, we’d write this function as

$$\text{fun } f \ x = f \ x$$

Note that we need to refer to $f$ in the body of $f$. One way we can achieve this in a language without built-in recursion is to pass the function to itself as its first argument. So we would have

$$\text{val } f' = \text{fn } f \Rightarrow \text{fn } x \Rightarrow f \ f \ x$$
$$\text{val } f = f' \ f'$$

This achieves general recursion, but if you typed it into SML/NJ’s REPL, it would not compile. This is because in a language like SML, this self-referential expression is not well-typed. This is evident from the fact that the $f'$ function immediately takes an argument of the same type as itself. The corresponding type must be infinite. However, in $\Lambda$, this does not matter. We can define

$$\lambda f. \lambda x .... f(x) ...$$

as we please.

However, writing this by hand is cumbersome, and so we create a function that performs this passing-function-to-itself operation for us. This is known as a fixed-point combinator. For example, the well-known $Y$ combinator performs this operation:

$$Y \triangleq \lambda F. (\lambda f. F(f \ f))(\lambda f. F(f \ f))$$

With $Y$, we can define $f$ in a more natural way instead (this time in $\lambda$-calculus):

$$f = Y(\lambda f. \lambda x. f \ x)$$

Why is this equivalent to the previous definition? Notice what happens when we $\beta$-convert this expression a bit, substituting $(\lambda f. \lambda x. f \ x)$ for $F$.

$$(\lambda f. (\lambda f. \lambda x. f x)(f \ f))(\lambda f. (\lambda f. \lambda x. f x)(f \ f)) =_{\beta} (\lambda f. (\lambda x. f \ f \ x))(\lambda f. (\lambda x. f \ f \ x))$$

What is $\lambda f. (\lambda x. f \ f \ x)$? It’s $f''$! When making a recursive call, the function passes itself into itself as its first argument. Now, to create $f$, all we need to do is apply $f'$ to itself, which is exactly what the above expression does.

This particular fixed-point combinator was discovered by Haskell Curry, and has the following property:

$$Y \ f = f(Y \ f) = f(f(Y \ f)) = \ldots$$

If we give $Y$ a self-referential function $f$, it produces an output which is equivalent to its own infinite iteration under $f$. Mathematically, this is known as a fixed point of $f$, an input which
is identical to its corresponding output. This construct allows us to create general recursive expressions. Notice especially how easy it is to introduce divergent (non-terminating) computation through this combinator. With $Y$, we can easily turn self-referential functions into recursive ones. An added advantage is the ease of defining $f$. Whereas before we had to apply the self-reference explicitly as in $f(f)$, this is no longer necessary with the $Y$ combinator; we may just write $f$.

The $Y$ combinator is not meant as a particularly practical method of writing recursive functions, nor is $\Lambda$ particularly practical as a programming language. However, it is a theoretically powerful construct that encodes recursion directly into the $\lambda$-calculus.