1 Lecture Plan

- Recursion and Induction
  - termination
  - Gauss formula \( \sum_{i=0}^{n} i = \frac{n \times (n+1)}{2} \)
  - evidence
- Tail recursion
- Sets and Types
  - disjoint union (variants)
  - match operator
  - see Hickey chapter 6, Kozen 2009 Lecture 4
2 Recursion and Induction

Recall that $\sigma(n) = \begin{cases} 0 & \text{if } n = 0 \\ \sigma(n - 1) + n & \text{else} \end{cases}$

In the Lecture 2 Notes we saw that

**Fact:** $\forall n : \mathbb{N}. \exists m : \mathbb{N}. (\sigma(n) = m)$

*Proof by induction on $n* 

- **Base:** $n = 0$, $\sigma(n) = 0$
- **Induction hypothesis:** $\exists m : \mathbb{N}. (\sigma(n - 1) = m)$
- Let $m_0$ be this number, so $\sigma(n - 1) = m_0$
- We must show $\exists m : \mathbb{N}. (\sigma(n) = m)$
  * By definition $\sigma(n) = \sigma(n - 1) + n$
    By the induction hypothesis we know the value of $\sigma(n - 1) = m_0$
    Thus by substituting in line * we known $\sigma(n) = m_0 + n$
    Thus the value $m$ we need is $m_0 + n$

Qed

This proof seems trivial, easy, elementary and obvious. So what does it demonstrate to us?

1. It shows that $\sigma(n)$ as a recursive function terminates. We can’t say this in OCaml as a general fact about $\mathbb{N}$.

2. In OCaml we can only know
   - $\sigma(0) \downarrow 0$
   - $\sigma(1) \downarrow 1$
   - $\sigma(2) \downarrow 3$
   - $\sigma(3) \downarrow 6$
   etc.

3. The inductive proof of termination is essentially building evidence recursively. The very method of induction on $\mathbb{N}$ is a claim about a whole family of recursive programs that follow this pattern.
• Base: build something (object, evidence, \ldots) for \( n = 0 \)

• Show how to take something built for value \( n - 1 \) and build it for \( n \)

\[ \text{object}(n - 1) \mapsto \text{object}(n) \text{ using some function } f \]

4. Induction is this recursive function

\[
\text{induction}(n) = \begin{cases} 
\text{if } n = 0 & \text{then evidence}(0) \\
\text{else } f(\text{induction}(n - 1)) 
\end{cases}
\]

In PS1 you saw how to view this as the iteration of function \( f \), namely

\[
f^{(n)}(\text{evidence}(0))
\]

where \[
\begin{cases} 
  f^{(0)}(a) = a \\
  f^{(n)}(a) = f(f^{(n-1)}(a)) 
\end{cases}
\]

Now let’s prove something much more interesting about \( \sigma(n) \) as \( \sum_{i=0}^{n} i \).

Gauss noticed this

\[
1 + 2 + 3 + \cdots + 99 + 100 \\
100 + 99 + 98 + \cdots + 2 + 1 \\
101 + 101 + 101 + \cdots + 101 + 101 = \frac{100\times(101)}{2}
\]

In general \( \sigma(n) = \frac{n(n+1)}{2} \).

We can prove this by induction and compute the sum at the same time.

**Theorem:** \( \forall n : \mathbb{N}. (\sigma(n) \ast 2 = n \ast (n + 1)) = \text{true} \)

**Proof by induction**

Base: \( \sigma(0) \ast 2 = 0 \ast (n + 1) = \text{true} \) since \( \sigma(0) = 0 \).

Induction: \textbf{assume} \( \sigma(n - 1) \ast 2 = (n - 1) \ast n \) to show \( \sigma(n) \ast 2 = n \ast (n + 1) \)
By definition
\[ \sigma(n) \times 2 = (n + \sigma(n-1)) \times 2 \]

By the induction hyp
\[ = (n \times 2 + \sigma(n-1) \times 2) \]
\[ = (n \times 2 + (n-1) \times n) \]
\[ = (n^2 + n) \]
\[ = n \times (n+1) \]
\[ = true \]

Qed

So the inductive proof is actually a recursive procedure we can execute.

The same proof we just gave can establish this more informative claim.

**Theorem:** \( \forall n : \mathbb{N}. \exists m : \mathbb{N}. \exists b : \text{Bool}. \)
\[ m = \sigma(n) \land \]
\[ b = (2 \times m = n \times (n+1)) \land \]
\[ b = true \]

The “program” we get from the proof is

\[ \text{induction}(n) = \text{if } n = 0 \text{ then } ((0, 2 \times 0 = 0 \times 1), true) \]
\[ \text{else } ((\sigma(n), 2 \times \sigma(n) = n \times n + 1), true) \]

Although recursive procedures in functions like \( \sigma \) are easy to reason about and understand, there are other good methods of doing these computations such as loops. Here is a simple generic while loop. It is slightly more efficient than \( \sigma \) as we see by performing the computations step by step.

\[
\begin{align*}
sum & := 0 \\
\text{while } n > 0 \text{ do} \\
& \quad sum := sum + n \\
& \quad n := n - 1 \\
\text{od}
\end{align*}
\]
\[ \{n = 0, sum = n + n - 1 + \cdots + 1\} \]

We can write this as a recursive function
\[
\text{while}(s, n) = \begin{cases} 
\text{if } n = 0 \text{ then } s \\
\text{else } \text{while}(s + n, n - 1) 
\end{cases}
\]

We can show that

\[
\text{while}(0, n) = \text{sigma}(n)
\]

We will see in due course how to write while loops generally as recursive functions. These are \textit{tail recursive functions}, a topic we study later. (OCaml won’t accept while as the name.)
3 Addendum

3.1 Sets and Types

In mathematics books, data is usually represented by sets, even for natural numbers $\mathbb{N}$.

- as a type: $0, 1, 2, 3, \ldots$
- as a set: $\emptyset$ empty set
  - $\{\emptyset\}$
  - $\{\emptyset, \{\emptyset\}\}$
  - $\vdots$

In mathematics, the union of sets is a primitive,

$$A \cup B = \{a’s, b’s\}$$
$$\mathbb{N} \cup \text{Bool} = \{true, false, 0, 1, 2, \ldots\}$$

In OCaml we use disjoint unions also called variant types or sum types.

```ocaml
type int_bool = Int of int | Bool of bool
```

3.2 Disjoint Union aka Variant types

How do we use elements of int_bool?

There is a new kind of expression

$$\text{match } e \text{ with } p_1 \to e_1 | \cdots | p_n \to e_n$$

Here is a function to convert int_bool to bool

```ocaml
let convert (x:int_bool):bool =
  match x with
    | Int (x) -> true
    | Bool (x) -> x
```