Goals

- Contrast type synonyms with new types
- See pattern-matching for built-in "one of" types (important for ML programming) and "each of" types
- Investigate why accumulator-style recursion can be more efficient

Type synonyms

You can bind a type name to a type. Example:

- type intpair = int * int
- type point = int * int
- type complex = int * int

(We call something else a type variable.)

In ML, this creates a synonym, also known as a transparent type definition. Recursion not allowed.

So a type name is equivalent to its definition.

To contrast, the type a datatype binding introduces is not equivalent to any other type (until possibly a later type binding).

Review: datatypes and pattern-matching

Evaluation rules for datatype bindings and case expressions:

datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn

Adds constructors Ci where Ci \( \nu \) is a value (and Ci has type \( \text{li} \rightarrow \text{li} \)).

\[
\text{case } \phi \text{ of } p1 \rightarrow \phi 1 | p2 \rightarrow \phi 2 | \ldots | pn \rightarrow \phi n
\]

- Evaluate \( \epsilon \) to \( \nu \)
- If \( \pi \) is the first pattern to match \( \nu \), then result is evaluation of \( \phi i \) in environment extended by the match.
- If \( \text{C} \) is a constructor of type \( t1 \ast \ldots \ast t\text{n} \rightarrow t \), then \( \text{C}(x1,\ldots,xm) \) is a pattern that matches \( \text{C}(\nu1,\ldots,\nu\text{n}) \) and the match extends the environment with \( x1 \) bound to \( \nu1 \) ... \( xm \) to \( \nu\text{n} \).
- Coming soon: many more pattern forms.
Why patterns?

Even without more pattern forms, this design has advantages over functions for “testing and deconstructing” (e.g., null, hd, and tl):

- easier to check for missing and redundant cases
- more concise syntax by combining “test, deconstruct, and bind”
- you can easily define testing and deconstructing in terms of pattern-matching

In fact, case expressions are the preferred way to test variants and extract values from all ML’s “one-of” types, including predefined ones (□ and :: just funny syntax).

So: Do not use functions hd, tl, null, isSome, val0f

Teaser: These functions are useful for passing as values

Tuple/record patterns

You can also use patterns to extract fields from tuples and records: pattern \{f1=x1, ..., fn=xn\} (or \(x1, ..., xn\)) matches \{f1=v1, ..., fn=vn\} (or \(v1, ..., vn\)).

For record-patterns, field-order does not matter.

This is better style than #1 and #foo, and it means you do not (ever) need to write function-argument types.

Instead of a case with one pattern, better style is a pattern directly in a \(\text{val}\) binding.

Next time: “deep” (i.e., nested) patterns.

Recursion

You should now have the hang of recursion:

- It’s no harder than using a loop (whatever that is)
- It’s much easier when you have multiple recursive calls (e.g., with functions over ropes or trees)

But there are idioms you should learn for elegance, efficiency, and understandability.

Today: using an accumulator.

Accumulator lessons

- Accumulators can avoid data-structure copying
- Accumulators can reduce the depth of recursive calls that are not tail calls
- Key idioms:
  - Non-accumulator: compute recursive results and combine
  - Accumulator: use recursive result as new accumulator
  - The base case becomes the initial accumulator

You will use recursion in non-functional languages—this lesson still applies.

Let’s investigate the evaluation of to_list_1 and to_list_2.