CSE 341: Programming Languages

Spring 2005
Lecture 4 — Mutation; “one-of” types; user-defined types
Where are we

- Done features: functions, tuples, lists, options, local bindings
- Done concepts: syntax vs. semantics, environments
- Today features: record types, datatypes, type synonyms, pattern-matching
- Today concepts: Mutation-free, “one-of” types, constructors/destructors, case-coverage
You want to *change* something?

There is no way to *mutate* (assign to) a binding, pair component, or list element.

How could the *lack* of a feature make programming easier?

In this case:

- Amount of sharing is indistinguishable
  - Aliasing irrelevant to correctness!

- Bindings are invariant across function application
  - Mutation breaks compositional reasoning, a (the?) intellectual tool of engineering
Base types and compound types

Languages typically provide a small number of “built-in” types and ways to build compound types out of simpler ones:

- Base types examples: `int`, `bool`
- Type builder examples: tuples, lists, `records`

Base types *clutter* a language definition; better to make them *libraries* when possible.

- ML does this to a remarkable extent (e.g., we will soon define away `bool` and conditionals)

Good to let programmers bind types to type names, just like we bind values to variables.
Compound-type flavors

Conceptually, just a few ways to build compound types:

1. “Each-of”: A t contains a t1 and a t2
2. “One-of”: A t contains a t1 or a t2
3. “Self-reference”: The definition of t may refer to t

Examples:

- int * bool
- int option
- int list

Remarkable: A lot of data can be described this way.
Convenient to think of as trees.

(optional) jargon: Product types, sum types, recursive types
User-defined types

There are many reasons to define your own types:

1. Using a tuple with 12 fields is incomprehensible

2. Writing down large types is unpleasant; we have computers for that

3. Large programs can use *abstract types* to be robust to change
   - A couple weeks ahead

4. So the language doesn’t have to “bake in” lists and options and…
Datatype

One-of types are less similar across languages

- We’ll discuss OO’s approach to one-of in a few weeks

In ML, we use make a new type with a datatype binding, e.g.:

datatype mytype = TwoInts of int*int
  | Str of string
  | Pizza

Semantics: Extend the environment with three constructors (in part, functions/constants that produce values of type mytype)

So we have a way to build them... what’s missing?
The old way

For lists, we had a way to:

- Test which \textit{variant} a value was

- Extract the values from \textit{value-carrying} variants
  - Makes no sense if you have the \textit{wrong} variant

What would this look like for \texttt{mytype}?
The new way

Rather than add \textit{variant-tests} and \textit{variant-destructors} (non-standard jargon and nothing to do with C++ destructors), ML has a \textit{case expression} that uses \textit{pattern-matching}.

In its simplest form, case has one pattern for each constructor in a datatype and binds one variable for each value carried. Example:

\begin{verbatim}
case e of
    TwoInts(i1,i2) => e1
  | Str s => e2
  | Pizza => e3
\end{verbatim}

What are the typing rules?
What are the evaluation rules?
Type-checking case

In addition to binding local variables and requiring branches to have the same type, the typing rules for case prevent some run-time errors:

- Exhaustiveness: No test can “fail” (a warning)
- Redundancy: No test can be “impossible” (an error)

So far, case gives us what we need to use datatypes:

- A (combined) way to test variants and extract values
- Powerful enough to define our own tests and destructors

In fact, pattern-matching is far more general and elegant:

- Can use it for datatypes already in the top-level environment
- Can use it for any type (later)
- Can have deep patterns (later)