Today's Lecture Notes for cs173

sk and dbtucker

October 16, 2000

Topic: Datatype

Datatype

Recall the Scheme datatype for *DSub* (we've added a *name* field to the *fresh-sub* variant just to make the example clearer):

(**define-datatype** *DSub DSub?*
  [*fresh-sub* (*name symbol?*)]
  [*new-sub* (*var symbol?*)
            (*value Value?*)
            (*rest DSub?*)])

How would you write the equivalent data structure in C? You need structs to to represent each variant...

```
struct {
  symbol *next;
} fresh_sub;

struct {
  symbol *var;
  Value *value;
  DSub *rest;
} new_sub;
```

...a union to combine the variants...

```
union {
  struct {
    symbol *next;
  } fresh_sub;
  struct {
    symbol *var;
    Value *value;
    DSub *rest;
```

```
  } new_sub;
} variants;
```

...and an outer struct to keep track of the variant type...

```
struct DSub {
  enum {FRESH_SUB, NEW_SUB} variant_type;
  union {
    struct {
      symbol *next;
    } fresh_sub;
    struct {
      symbol *var;
      Value *value;
      DSub *rest;
    } new_sub;
  } variants;
};
```

Note that in the C version, we need to use a pointer (`DSub *`) to represent a recursive data structure. However, when we program over recursive data structures, we use recursive functions, not goto's. The analogy between pointers and goto's is strong; they are the analogous constructs in the data and control domains.

Pointers are the goto's of data-structures.

Just as we don't use unconstrained goto's in our programs, we shouldn't in our data either. Indeed, pointers are even more insidious, because most languages won't allow you to add something to a goto label to get a new label, which you can then use as the target of a jump. Yet you can do that with pointers—and since C has function pointers, you can simulate the effect of "goto label addition" and jump right into the middle of nonsense.

The Maybe datatype

Remember that the type rule for `if` requires that both branches have the same type:

$$\frac{\Gamma \vdash c : \texttt{bool} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash f : \tau}{\Gamma \vdash (\texttt{if}\ \ c\ \ t\ \ f) : \tau}$$

This means we can't write expressions like:

```
(if ...
    true
    3)
```

What use is such an expression, though? One example is Scheme's *member* function; (*member x L*) returns #f is x is not in the list L, otherwise it returns the sublist beginning with x.

However, this function wouldn't type check if Scheme used the above type rule for **if**. To make both branches return the same type, we could create a new datatype called *Bool_Or_List*:

(**define-datatype** *Bool_Or_List Bool_Or_List?*
  [*aBool* (*b boolean?*)]
  [*aList* (*l list?*)])

Then the function would return (*aBool* #f) or (*aList l*) depending on the case, and the type checker would determine that the return type of the function is *Bool_Or_List*.

A more general datatype for expressing such values is the *Maybe* datatype:

(**define-datatype** *Maybe Maybe?*
  [*yes* (*v value?*)]
  [*no*])

The idea is to use the *yes* variant when returning some value, and the *no* variant when there is no value. For example, the *member* function would return (*yes l*) if the element was in the list, or (*no*) otherwise.

A similar datatype can be used for representing file ports. At any time, a file port is in one of three states: unopened, open, or closed. The datatype reflects these states:

(**define-datatype** *FilePort FilePort?*
  [*unopened*]
  [*open* (*p Port?*)]
  [*closed*])

Adding datatype to our language

We want the ability to define new datatypes in our language. We'll add a `datatype` expression, which defines the new type and allows it to be used in the body of the expression. The grammar is:

$$
\begin{array}{rcl}
L & ::= & \ldots \\
 & | & (\texttt{datatype}\ \tau \\
 & & \quad [var_1\ \ (field_{11}\ \ \tau)\ \ (field_{12}\ \ \tau)\ \ldots] \\
 & & \quad [var_2\ \ (field_{21}\ \ \tau)\ \ (field_{22}\ \ \tau)\ \ldots] \\
 & & \quad \langle L \rangle\ )
\end{array}
$$

For example, we can use `datatype` to create number lists:

```
(datatype nlist
  [mt]
  [cons (first num) (rest nlist)]
```

...)

The `datatype` expression creates the new type specified by τ, and binds several new functions in the body:

Contructors The functions var_1 and var_2 are constructors. For example, var_1 takes $field_{11}$, $field_{12}$, &c. as arguments, and returns a new value of type τ.

Predicates The functions var_1? and var_2? are predicates. For example, var_1? takes a value of type τ and returns `true` iff the value was created by var_1.

Selectors (or field accessors) The functions $field_{11}$, $field_{21}$, &c. are selectors. For example, $field_{11}$ takes a value of type τ, and returns the value denoted by $field_{11}$. Note that ($field_{11}$ (var_2 v)) should raise an exception, because the function $field_{11}$ only operates on values which were created by var_1.

To summarize, in the body of a `datatype` expression, there is a new type τ, and new functions for creating and using values of this type—constructors, predicates, and selectors. In the `nlist` example, the new functions are:

- constructors: `mt` and `cons`
- predicates: `mt?` and `cons?`
- selectors: `first` and `rest`