Dynamic Datatype-Programming

OCAML's ability to create new datatypes via type variants and products is extremely useful. It is also partly forced on us by the typing restrictions, as we can't otherwise have a function that can take both integers and floats and strings etc. So in Racket we can have a function that does one thing if its input is a pair, another thing if the input is a number and so on. Racket offers us predicates like pair?, number? etc to be able to do that.

We will see now how to create our own datatypes, in two ways.

Dynamic Datatype-Programming via tagged lists

We can create our own predicates and "constructors". The idea is simple, using "tagged lists":

- We represent each "value" as a list.
- The first entry holds a symbol indicating the kind/type of value.
- The remaining entries hold various components of the value.
- We build a "constructor" function that creates the cons cell.
- We build a predicate that tests if the symbol in the first entry is the right one.
- We build accessors to get hold of the other entries.

This is in fact not unlike how these kinds of data structures are actually implemented in memory.

For example, suppose we wanted to have something equivalent to the expression datatype we've been working with, a very simplified form of which would be:

```
type expr = Num of i
          | Add of expr * expr
          | Mult of expr * expr
```

Here is how that might look in Racket, with our plan to use lists to represent different datatypes:

```
;; Constructors
(define (Num i) (list 'Num i))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Mult e1 e2) (list 'Mult e1 e2))
;; Predicates
(define (Num? e) (eq? (car e) 'Num))
(define (Add? e) (eq? (car e) 'Add))
(define (Mult? e) (eq? (car e) 'Mult))
;; Accessors
(define (Num-i e) (car (cdr e)))
(define (Add-e1 e) (car (cdr e)))
```

```
(define (Add-e2 e) (car (cdr (cdr e))))
(define (Mult-e1 e) (car (cdr e)))
(define (Mult-e2 e) (car (cdr (cdr e))))
```

With these definitions in mind, here's how the evaluation of an expression might go:

```
(define (eval-expr e)
  (cond [(Num? e) (Num-i e)]
        [(Add? e) (+ (eval-expr (Add-e1 e))
                     (eval-expr (Add-e2 e)))]
        [(Mult? e) (* (eval-expr (Mult-e1 e))
                      (eval-expr (Mult-e2 e)))]
        [else (error "Unknown expression")]))

(eval-expr (Num 4))
(eval-expr (Add (Num 4)
                (Mult (Num 3) (Num 5))))
```

As you can imagine, writing all the accessors above can be tedious. Racket in fact offers us an alternative, in the form of structs. You could also imagine some macro mechanism that automatically does the kind of work needed to create the above definitions. In fact that's what structs more or less do for us, with some extra features added in.

Before we move on though, let us discuss some of the problems with the previous approach:

- There are not many checks anywhere to make sure the input is right. For example, any list, produced anywhere, whose first entry is the symbol `'Num` can pass the `Num?` predicate.
- We don't have any way to ensure that these lists have the right number of elements. Though we could in theory make our predicates a bit smarter.
- The lists are exposed as such. We don't have a way to hide their nature from other parts of the program. For example any of our "structures" would pass the `list?` predicate.
- It's not easy in general to test whether something is an arithmetic expression. We could create a test like so: `racket (define (expr? e) (or (Num? e) (Add? e) (Mult? e)))` But this already has too many false positives as we described above, and we will need to remember to update it every time we add a new expression case.

We will see how structs remedy a lot of these "problems".

Dynamic Datatype-Programming via structs

The `struct` constructs for our example above would look as follows, where we switch to lowercase first letter (do not worry about the `#:transparent` flag right now, we may discuss it later):

```
(struct num (i) #:transparent)
(struct add (e1 e2) #:transparent)
(struct mult (e1 e2) #:transparent)
```

These three lines alone essentially create all the functions we defined explicitly earlier:

- We can use `(num 3)` or `(add (num 3) (num 4))` and so on to create new "expressions". It's not perfect as it would allow us for instance to write `(add 2 3)`, i.e. it doesn't enforce a "type" for the components. We may see a way around that when we discuss contracts.
- We have predicates `num?`, `add?` etc: `(num? (num 3))`.
- We have accessors: `(add-e1 (add (num 3) (num 4)))` gives us `(num 3)`.
- If we had added the flag `#:mutable` then we would also have been given `set!`-style functions for each component, if that is something we felt would be useful.

The expression evaluator will look very similar to the one we wrote earlier. The main difference is that while before we could have used the fact that expression are actually lists to directly access their entries, with structs we are forced to use the accessor methods, which is a very good thing.