PL: Lecture #5

Tuesday, September 24th

Intro to Typed Racket

The plan:

- Why Types?
- Why Typed Racket?
- What’s Different about Typed Racket?
- Some Examples of Typed Racket for Course Programs

Types

- Who has used a (statically) typed language?
- Who has used a typed language that’s not Java?

Typed Racket will be both similar to and very different from anything you’ve seen before.

Why types?

- Types help structure programs.
- Types provide enforced and mandatory documentation.
- Types help catch errors.

Types ***will*** help you. A *lot*.

Structuring programs

- Data definitions

```
;; An AE is one of:   ; \
;;  (make-Num Number) ;  > HtDP
;;  (make-Add AE AE)  ; /

(define-type AE       ; \
  [Num number?]       ;  > Predicates =~= contracts (PLAI)
  [Add AE? AE?])      ; /    (has names of defined types too)

(define-type AE       ; \
  [Num Number]        ;  > Typed Racket (our PL)
  [Add AE AE])        ; /
```

- Data-first

  The structure of your program is derived from the structure of your data.

  You have seen this in Fundamentals with the design recipe and with templates. In this class, we will see it extensively with type definitions and the (cases …) form. Types make this pervasive — we have to think about our data before our code.

- A language for describing data

Instead of having an informal language for describing types in contract lines, and a more formal description of predicates in a `define-type` form, we will have a single, unified language for both of these. Having such a language means that we get to be more precise and more expressive (since the typed language covers cases that you would otherwise dismiss with some hand waving, like "a function").

Why Typed Racket?

Racket is the language we all know, and it has the benefits that we discussed earlier. Mainly, it is an excellent language for experimenting with programming languages.

- Typed Racket allows us to take our Racket programs and typecheck them, so we get the benefits of a statically typed language.
- Types are an important programming language feature; Typed Racket will help us understand them.

[Also: the development of Typed Racket is happening here in Northeastern, and will benefit from your feedback.]

How is Typed Racket different from Racket

- Typed Racket will reject your program if there are type errors! This means that it does that at compile-time, *before* any code gets to run.
- Typed Racket files start like this:

  ```
  #lang typed/racket
  ;; Program goes here.
  ```

  but we will use a variant of the Typed Racket language, which has a few additional constructs:

  ```
  #lang pl
  ;; Program goes here.
  ```

- Typed Racket requires you to write the contracts on your functions.

  Racket:

  ```
  ;; f : Number -> Number
  (define (f x)
    (* x (+ x 1)))
  ```

  Typed Racket:

  ```
  #lang pl
  (: f : Number -> Number)
  (define (f x)
    (* x (+ x 1)))
  ```

  [In the "real" Typed Racket the preferred style is with prefix arrows:

  ```
  #lang typed/racket
  (: f (-> Number Number))
  (define (f x) : Number
    (* x (+ x 1)))
  ```

  and you can also have the type annotations appear inside the definition:

  ```
  #lang typed/racket
  (define (f [x : Number]) : Number
    (* x (+ x 1)))
  ```

  but we will not use these form.]

- As we’ve seen, Typed Racket uses types, not predicates, in `define-type`.

```
(define-type AE
  [Num Number]
  [Add AE AE])
```

  versus

```
(define-type AE
  [Num number?]
  [Add AE? AE?])
```

- There are other differences, but these will suffice for now.

Examples

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [else        "a lot"]))

(: fact : Number -> Number)
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))

(: helper : Number Number -> Number)
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))
(: fact : Number -> Number)
(define (fact n)
  (helper n 1))

(: fact : Number -> Number)
(define (fact n)
  (: helper : Number Number -> Number)
  (define (helper n acc)
    (if (zero? n)
      acc
      (helper (- n 1) (* acc n))))
  (helper n 1))

(: every? : (All (A) (A -> Boolean) (Listof A) -> Boolean))
;; Returns false if any element of lst fails the given pred,
;; true if all pass pred.
(define (every? pred lst)
  (or (null? lst)
      (and (pred (first lst))
           (every? pred (rest lst)))))

(define-type AE
  [Num Number]
```

```
    [Add AE AE]
    [Sub AE AE])

  ;; the only difference in the following definition is
  ;; using (: <name> : <type>) instead of ";; <name> : <type>"

  (: parse-sexpr : Sexpr -> AE)
  ;; parses s-expressions into AEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n) (Num n)]
      [(list '+ left right)
       (Add (parse-sexpr left) (parse-sexpr right))]
      [(list '- left right)
       (Sub (parse-sexpr left) (parse-sexpr right))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

More interesting examples

- Typed Racket is designed to be a language that is friendly to the kind of programs that people write in Racket. For example, it has unions:

  ```
  (: foo : (U String Number) -> Number)
  (define (foo x)
    (if (string? x)
      (string-length x)
      ;; at this point it knows that `x' is not a
      ;; string, therefore it must be a number
      (+ 1 x)))
  ```

  This is not common in statically typed languages, which are usually limited to only *disjoint* unions. For example, in OCaml you'd write this definition:

  ```
  type string_or_number = Str of string | Int of int ;;
  let foo x = match x with Str s -> String.length s
                         | Int i -> i+1 ;;
  ```

  And use it with an explicit constructor:

  ```
  foo (Str "bar") ;;
  foo (Int 3) ;;
  ```

- Note that in the Typed Racket case, the language keeps track of information that is gathered via predicates — which is why it knows that one `x` is a String, and the other is a Number.
- Typed Racket has a concept of subtypes — which is also something that most statically typed languages lack. In fact, the fact that it has (arbitrary) unions means that it must have subtypes too, since a type is always a subtype of a union that contains this type.
- Another result of this feature is that there is an `Any` type that is the union of all other types. Note that you can always use this type since everything is in it — but it gives you the *least* information about a value. In other words, Typed Racket gives you a choice: *you* decide which type to use, one that is very restricted but has a lot of information about its values to a type that is very permissive but has almost no useful information. This is in contrast to other type system (HM systems) where there is always exactly one correct type.

  To demonstrate, consider the identity function:

  ```
  (define (id x) x)
  ```

You could use a type of `(: id : Integer -> Integer)` which is very restricted, but you know that the function always returns an integer value.

Or you can make it very permissive with a `(: id : Any -> Any)`, but then you know nothing about the result — in fact, `(+ 1 (id 2))` will throw a type error. It *does* return `2`, as expected, but the type checker doesn't know the type of that `2`. If you wanted to use this type, you'd need to check that the result is a number, eg:

```
(let ([x (id 123)]) (if (number? x) (+ x 10) 999))
```

This means that for this particular function there is no good *specific* type that we can choose — but there are *polymorphic* types. These types allow propagating their input type(s) to their output type. In this case, it's a simple "my output type is the same as my input type":

```
(: id : (All (A) A -> A))
```

This makes the output preserve the same level of information that you had on its input.

- Another interesting thing to look at is the type of `error`: it's a function that returns a type of `Nothing` — a type that is the same as an *empty* union: `(U)`. It's a type that has no values in it — it fits `error` because it *is* a function that doesn't return any value, in fact, it doesn't return at all. In addition, it means that an `error` expression can be used anywhere you want because it is a subtype of anything at all.
- An `else` clause in a `cond` expression is almost always needed, for example:

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)    1]
        [(<= n 99)   2]
        [(<= n 999)  3]
        [(<= n 9999) 4]
        [(>  n 9999) "a lot"]))
```

(and if you think that the type checker should know what this is doing, then how about

```
(> (* n 10) (/ (* (- 10000 1) 20) 2))
```

or

```
(>= n 10000)
```

for the last test?)

- In some rare cases you will run into one limitation of Typed Racket: it is difficult (that is: a generic solution is not known at the moment) to do the right inference when polymorphic functions are passed around to higher-order functions. For example:

```
(: call : (All (A B) (A -> B) A -> B))
(define (call f x)
  (f x))
(call rest (list 4))
```

In such cases, we can use `inst` to *instantiate* a function with a polymorphic type to a given type — in this case, we can use it to make it treat `rest` as a function that is specific for numeric lists:

```
(call (inst rest Number) (list 4))
```

In other rare cases, Typed Racket will infer a type that is not suitable for us — there is another form, `ann`, that allows us to specify a certain type. Using this in the `call` example is more verbose:

```
(call (ann rest : ((Listof Number) -> (Listof Number))) (list 4))
```

However, these are going to be rare and will be mentioned explicitly whenever they're needed.

Bindings & Substitution

We now get to an important concept: substitution.

Even in our simple language, we encounter repeated expressions. For example, if we want to compute the square of some expression:

```
{* {+ 4 2} {+ 4 2}}
```

Why would we want to get rid of the repeated sub-expression?

- It introduces a redundant computation. In this example, we want to avoid computing the same sub-expression a second time.
- It makes the computation more complicated than it could be without the repetition. Compare the above with:

```
with x = {+ 4 2},
  {* x x}
```

- This is related to a basic fact in programming that we have already discussed: duplicating information is always a bad thing. Among other bad consequences, it can even lead to bugs that could not happen if we wouldn't duplicate code. A toy example is "fixing" one of the numbers in one expression and forgetting to fix the corresponding one:

  ```
  {* {+ 4 2} {+ 4 1}}
  ```

  Real world examples involve much more code, which make such bugs very difficult to find, but they still follow the same principle.
- This gives us more expressive power — we don't just say that we want to multiply two expressions that both happen to be `{+ 4 2}`, we say that we multiply the `{+ 4 2}` expression by *itself*. It allows us to express identity of two values as well as using two values that happen to be the same.

So, the normal way to avoid redundancy is to introduce an identifier. Even when we speak, we might say: "let x be 4 plus 2, multiply x by x".

(These are often called "variables", but we will try to avoid this name: what if the identifier does not change (vary)?)

To get this, we introduce a new form into our language:

```
{with {x {+ 4 2}}
  {* x x}}
```

We expect to be able to reduce this to:

```
{* 6 6}
```

by substituting 6 for `x` in the body sub-expression of `with`.

A little more complicated example:

```
{with {x {+ 4 2}}
  {with {y {* x x}}
    {+ y y}}}
[add]  = {with {x 6} {with {y {* x x}} {+ y y}}}
[subst]= {with {y {* 6 6}} {+ y y}}
[mul]  = {with {y 36} {+ y y}}
[subst]= {+ 36 36}
[add]  = 72
```

WAE: Adding Bindings to AE

> ☞ *PLAI §3*

To add this to our language, we start with the BNF. We now call our language “WAE” (With+AE):

```
<WAE> ::= <num>
        | { + <WAE> <WAE> }
        | { - <WAE> <WAE> }
        | { * <WAE> <WAE> }
        | { / <WAE> <WAE> }
        | { with { <id> <WAE> } <WAE> }
        | <id>
```

Note that we had to introduce *two* new rules: one for introducing an identifier, and one for using it. This is common in many language specifications, for example `define-type` introduces a new type, and it comes with `cases` that allows us to destruct its instances.

For `<id>` we need to use some form of identifiers, the natural choice in Racket is to use symbols. We can therefore write the corresponding type definition:

```
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])
```

The parser is easily extended to produce these syntax objects:

```
(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(list 'with (list (symbol: name) named) body)
     (With name (parse-sexpr named) (parse-sexpr body))]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

But note that this parser is inconvenient — if any of these expressions:

```
{* 1 2 3}
{foo 5 6}
{with x 5 {* x 8}}
{with {5 x} {* x 8}}
```

would result in a “bad syntax” error, which is not very helpful. To make things better, we can add another case for `with` expressions that are malformed, and give a more specific message in that case:

```
(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(list 'with (list (symbol: name) named) body)
     (With name (parse-sexpr named) (parse-sexpr body))]
    [(cons 'with more)
     (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

and finally, to group all of the parsing code that deals with `with` expressions (both valid and invalid ones), we can use a single case for both of them:

```
(: parse-sexpr : Sexpr -> WAE)
;; parses s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     ;; go in here for all sexpr that begin with a 'with
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

And now we're done with the syntactic part of the `with` extension.

> *Quick note — why would we indent `With` like a normal function in code like this*
>
> ```
> (With 'x
>       (Num 2)
>       (Add (Id 'x) (Num 4)))
> ```
>
> *instead of an indentation that looks like a `let`*
>
> ```
> (With 'x (Num 2)
>   (Add (Id 'x) (Num 4)))
> ```
>
> *?*
>
> *The reason for this is that the second indentation looks like a binding construct (eg, the indentation used in a `let` expression), but `With` is not a binding form — it's a plain function because it's at the Racket level. You should therefore keep in mind the huge difference between that `With` and the `with` that appears in WAE programs:*
>
> ```
> {with {x 2}
>   {+ x 4}}
> ```
>
> *Another way to look at it: imagine that we intend for the language to be used by Spanish/Chinese/German/French speakers. In this case we would translate "`with`":*

> ```
> {con  {x 2} {+ x 4}}
> {he   {x 2} {+ x 4}}
> {mit  {x 2} {+ x 4}}
> {avec {x 2} {+ x 4}}
> {c    {x 2} {+ x 4}}
> ```
>
> *but we will not do the same for `With` if we (the language implementors) are English speakers.*