Lecture notes:
Functional Programming in Typed Racket

Florian Biermann
fbie@itu.dk

2016-05-26

1 Hello World in Typed Racket

```
;; This is a comment!
;; Tell the run-time, which language to use.
#lang typed/racket

;; Now, print something.
(print "Nihao!")
```

You must always start your code with a `#lang` directive. You can also choose other languages than `typed/racket`, such as `racket`, `plaid` or `slides`. For readability, we will omit all `#lang` directives in the remaining code examples.

2 Expressions

Expressions in Typed Racket are of the form:

`(f arg1 arg2 ... argn)`

Operators are also just functions:

`(+ x y)`	$\Rightarrow$	$x + y$
`(> x y)`	$\Rightarrow$	$x > y$
`(/ x y)`	$\Rightarrow$	$\frac{x}{y}$
`(f x y)`	$\Rightarrow$	$f(x, y)$

In Racket this is called an *S-expression*.

3 Local Bindings

Local bindings are very similar to local bindings in for instance Java. However, their scope is very explicit:

```
(let ([x (* 2 16)]
      [y (* 3 17)])
    (print (+ x y)))
```

Note that, after the final parenthesis that closes the `let` expression, you cannot reference `x` or `y`:

```
(let ([x (* 2 16)]
      [y (* 3 17)])
    (print (+ x y)))
(print x)
```

Trying this results in an error:

```
; stdin::5873:
;  Type Checker: missing type for top-level identifier;
;  either undefined or missing a type annotation
;   identifier: x
;   in: x
```

4 Functions and Types

We'd like to define a simple function `times-two` that multiplies its argument `x` by two:

```
(: times-two (-> Number Number))
(define (times-two x)
  (* 2 x))
```

There are two things to note here. The first is that there is no return statement in Racket. Instead, a function evaluates to the result of the last expression in its body. The second is that `:` denotes a *type annotation*. We have annotated `times-two` to be of type $Number \to Number$.

In a type-expression, the return-type is the last type that occurs. Consider the type `(-> A B C)`. The return-type is `C` and `A` and `B` are parameter types. If the type is `(-> A B A)`, `A` is both parameter and return type. `B` is only parameter type.

We could also have written `times-two`'s type like this:

```
(define (times-two [x : Number])
  (* 2 x))
```

Typed Racket can then infer the correct type of the function. We can ask the REPL[1] for the `times-two`'s type:

```
> times-two
- : (-> Number Number)
#<procedure:times-two>
```

[1]REPL is short for Read-Eval-Print-Loop.

We will often use both notations. The latter notations is in particular used to define the parameter types for `struct` constructors.

5 Conditionals

This is a well-known function:

```
(: abs (-> Number Number))
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

(Exercise: How else could you have written the type?)

It contains a conditional. Conditionals have the form `(if B ET EF)` where `B` is a boolean expression, `ET` is the expression that is evaluated if `B` evaluates to true (`#t`), `EF` is the expression that is evaluated if `B` evaluates to false (`#f`).

In the lecture, I told you that the types of `ET` and `EF` must be the same. This is true in most functional programming languages. However, Typed Racket is a bit special. Look at this example:

```
(define (funny [b : Boolean])
  (if b "True" 54727565))
```

Let's check the function's type:

```
> funny
- : (-> Boolean (U Positive-Index String))
#<procedure:funny>
```

Instead of raising a type-error, Typed Racket infers from the usage of the conditional, that the return-type of `funny` must be the union of the types of both branches. `U` stands for set-union, $\bigcup$.

Typed Racket features polymorphic types. Polymorphic types in Typed Racket are very explicit:

```
(: twice (All (A) (-> A (Pairof A A))))
(define (twice a)
  (cons a a))
```

`cons` is the constructor for the `Pairof` type. You can read the type annotation as "for all types `A`, the type of `twice` is such that iff you pass it a value of some type `A` it will return a pair of type `A` $\times$ `A`".

6 Immutable State and Recursion

In purely functional languages, you cannot alter the state of the program. You cannot assign to a variable once it has been defined[2]. Consider this simple Java

[2]Actually, this is possible in Racket, but we pretend that it isn't because it makes a lot of things so much harder.

program:

```
public static bool isEven(int n) {
  while (1 < n)
    n = n - 2;
  return n == 0;
}
```

The function decrements `n` until it is less or equal to one. Then, it check whether `n` is equal to zero and returns the value of that check. While this is a very simple way to do it, because recursion in Java is costly, we cannot do it that way in Racket, because there, `n` can never be changed. Instead, we have to chose the recursive approach:

```
(: is-even? (-> Integer Boolean))
(define (is-even? n)
  (if (< 1 n)
      (is-even? (- n 2))
      (= n 0)))
```

Every time, `is-even?` is called with an `n` that is greater than 1, it calls itself with `(- n 2)`. The result is the same, but this way of thinking enables us to decompose a function into its *cases*!

7 Pattern Matching

Pattern matching is a very powerful tool of functional programming. We will use it time and time again to implement functions. Here is a variant of `is-even?` that uses pattern matching:

```
(: is-even? (-> Integer Boolean))
(define (is-even? n)
  (match n
    [0 #t]
    [1 #f]
    [_ (is-even? (- n 2))]))
```

The principle of the function stays the same: if the value is zero, then it is an even number. If it is one, it is an odd number. In any other case (this is what `_` means), the function calls itself recursively and decrements `n` by two. The wildcard `_` matches any value and it must always come last in the list of patterns, otherwise it will take over all following patterns, too.

By the way, you can use `_` anywhere you like if you do not care about the value of something.

8 Anonymous Functions

An anonymous function is a function without a name. You can create it and apply it immediately, as in this example:

```
> ((lambda (x) x) 2)
2
```

Here, we have defined a lambda expression and applied it to the value 2. We call this function the *identity function*. The line above is equal to $((\lambda\, x\,.\, x)\, 2)$.

9 The Maybe Type

The `Maybe` type is useful to model the absence of a value without having to explicitly check for null-pointers:

```
(struct None ())
(struct (A) Some ([a : A]))
(define-type (Maybe A) (U None (Some A)))
```

The type of `Maybe` is the union of two structs and it is polymorphic! `A` is just a placeholder, a generic name for a type. It is simlilar, but not equal, to this Java code:

```
public static abstract class Maybe<A> {}

public static class None<A> extends Maybe<A> {
  public None() {}
}

public static class Some<A> extends Maybe<A> {
  public final A a;
  public Some(A a) {
    this.a = a;
  }
}
```

Now, we can implement a function that performs safe division:

```
(: maybe-divide (-> Number Number (Maybe Number)))
(define (maybe-divide n d)
  (if (= d 0)
      (None)
      (Some (/ n d))))
```

We first check whether the divisor `d` is zero. If yes, then we cannot divide, so we return `None`. Otherwise, we return `Some (\ n d)`. This has many benefits. For the first, we do not need to catch any exceptions. Exceptions can go unnoticed when you write your code and then it later becomes a problem in production.

Secondly, the type of `maybe-divide` indicates that the computation can go wrong! This is useful information if you are writing an API and want to tell the user of the API that there is a chance for this function to fail. This might be anything from simple arithmetic to HTTP-requests.

We can check whether a `Maybe` has a value:

```
(: has-value? (-> (Maybe Any) Boolean))
(define (has-value? m)
  (match m
    [(None) #f]
    [(Some _) #t]))
```

We use pattern matching do *decompose* the value of `m` in order to see how it has been constructed and what is inside. Note that we do not care what value it contains, so we use `_`.

We can use a *higher-order function* – that is a function that takes another function as an argument – to only execute a function on a `Maybe` if it contains a value:

```
(: maybe-map (All (A B) (-> (-> A B)
                            (Maybe A)
                            (Maybe B))))
(define (maybe-map f m)
  (match m
    [(None) (None)]
    [(Some a) (Some (f a))]))
```

`maybe-map` is a polymorphic function, because it does not care what the *actual type* of `A` and `B` is.

`f` is of type `(-> A B)`. This is the function we want to apply if there is a value present in the maybe. It is important that the types match. Because `f` accepts a type `A` and returns a type `B`, the type of the second parameter to `maybe-map` must be of type `(Maybe A)` and the return-type must be of type `(Maybe B)`.

Now, we can use `maybe-map` together with a *lambda expression*:

```
> (maybe-map (lambda ([x : Number]) (* 2 x))
             (maybe-divide 42 23))
- : (U None (Some Number))
(Some 84/23)
```

10 Single-Linked Lists

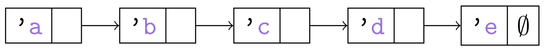
Let us try to implement a more interesting data type. Lists are very useful, so let us implement the simplest list possible, a single-linked list. We need an empty list constructor, which we call `Nil` and a constructor for adding an element to a list, which we call `Cons`:

```
(struct Nil ())
(struct (A) Cons ([head : A] [tail : (LinkedList A)]))
(define-type (LinkedList A) (U Nil (Cons A)))
```

Let us instantiate such a list:

```
(Cons 'a
      (Cons 'b
            (Cons 'c
                  (Cons 'd
                        (Cons 'e (Nil))))))
```

This is how the list looks like, conceptually:



Every list instance contains another, smaller list instance. We call the value of a Cons cell head and the remaining list tail.

Let us see how we can compute the length of a list. We can use pattern-matching:

```
(: list/length (All (A) (-> (LinkedList A) Integer)))
(define (list/length as)
  (match as
    [(Nil) 0]
    [(Cons _ tail) (+ 1 (list/length tail))]))
```

There are two cases: either, the list is empty. In this case, the length is zero. Or the list has at least one element. Then, the length is one plus the length of the tail. Again, note that we do not care about the value of the Cons cell.

We can easily implement a function to push a new element to the list:

```
(: list/push (All (A) (-> A (LinkedList A)
                            (LinkedList A))))
(define (list/push a as)
  (Cons a as))
```

It takes constant time and space, because we re-use the old list. Indeed, many lists can share the same sub-list. For instance, all lists share the empty list Nil. This is legal, because lists are also immutable[3]!

Let us look at more functions on lists. For example, we would like to be able to concatenate two lists. We can again implement this conveniently by pattern-matching.

```
(: list/concat (All (A) (-> (LinkedList A)
                            (LinkedList A)
                            (LinkedList A))))
(define (list/concat lhs rhs)
```

[3]Sometimes people also call cons-lists a *persistent data structure*.

```
  (match lhs
    [(Nil) rhs]
    [(Cons head tail) (Cons head
                            (list/concat tail rhs))]))
```

Again, we perform case analysis: if the left hand side `lhs` is empty, then there is nothing to do and returning whatever is on the right hand side, `rhs`, is correct. If however `lhs` is not empty, we must add its `head` to the new list. How do we construct a new list? By calling `list/concat` on its `tail`.

Why is this correct? Because either, `tail` is `Nil`. In this case, `concat` will return `rhs`. So we have added `head` to `rhs`. This is, what we wanted. Or, `tail` is not empty. Then we add its current `head` to whatever is returned by `concat` on its `tail`. At some point, there is no `tail` left, because lists cannot be infinite (computers have finite memory), and we end up again at the base case.

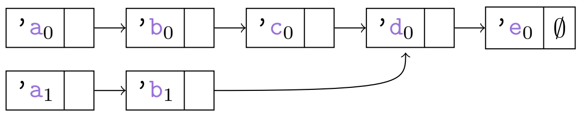
Note, that this takes $O(n)$ time and space, where n is the length of `lhs`. We have to look at every element once but we also have to construct n new cells, because we cannot simply replace the empty list at the very end of `lhs` by `rhs`: lists are also immutable!

The `list/remove` function is a nice example to illustrate sub-list sharing. If we know that an element is inside the list, we can ask `list/remove` to remove it:

```
> (list/remove
     'c (Cons 'a
              (Cons 'b
                    (Cons 'c
                          (Cons 'd
                                (Cons 'e (Nil)))))))

(Cons 'a (Cons 'b (Cons 'd (Cons 'e #<Nil>))))
```

How does it do that? It simply traverses the list and builds new `Cons` cells until it encounters `'c`. Then it skips `'c` and returns the remaining list:



This is the code:

```
(: list/remove (All (A) (-> A (LinkedList A)
                            (LinkedList A))))
(define (list/remove a as)
  (match as
    [(Nil) (Nil)]
    [(Cons head tail) (if (eq? head a)
                           tail
                           (Cons head
                                 (list/remove a tail)))]))
```

11 Higher-Order Functions on Lists

```
(: list/map (All (A B) (-> (-> A B)
                           (LinkedList A)
                           (LinkedList B))))
(define (list/map f as)
  (match as
    [(Nil) (Nil)]
    [(Cons head tail) (Cons (f head)
                            (list/map f tail))]))

(: list/filter (All (A) (-> (-> A Boolean)
                            (LinkedList A)
                            (LinkedList A))))
(define (list/filter p as)
  (match as
    [(Nil) (Nil)]
    [(Cons head tail) (if (p head)
                          (list/filter p tail)
                          (Cons head
                                (list/filter p tail)))]))

(: list/fold (All (A B) (-> (-> B A B)
                           B
                           (LinkedList A) B)))
(define (list/fold f state as)
  (match as
    [(Nil) state]
    [(Cons head tail) (list/fold f
                                 (f state head)
                                 tail)]))
```