# Programming with Refinement Types in Typed Racket

## Introduction

Code from chapters 2 through 4 of Programming with Refinement Types translated from Liquid Haskell to Typed Racket. Programming with Refinement Types is written by Professor Ranjit Jhala at the University of California, San Diego.

This translation retains the function and type names of the original document where possible.

As the Liquid Haskell and Typed Racket compilers have different features as well as different logics I have had to use additional type annotations and helper functions to get the code to compile while keeping it true to the spirit of the original document.

## Preliminaries

To ensure you have the right language loaded, start your Racket file with `#lang typed/racket #:with-refinements`

I wanted to use `Int` instead of `Integer` to make my version of the code cosmetically resemble the original so I added `(define-type Int Integer)` just after the line with `#lang`.

I will override existing Racket functions and types like `Zero` and `List` over the course of this document to match the original one. I hope it isn't confusing.

## Simple Refinement Types

### Types, Predicates, Expressions

The grammar for refinement types in Typed Racket is different from that of Liquid Haskell. It is stated in the Typed Racket Reference under the section Logical Refinements and Linear Integer Reasoning

#### Example: Integers equal to 0

```
(define-type Zero (Refine (v : Int) (= v 0)))

(define zero : Zero 0)
```

#### Example: Natural Numbers

```
(define-type Nat (Refine (v : Int) (<= 0 v)))

(define nats : (Listof Nat) (list 0 1 2 3))
```

#### Exercise: Positive Integers

Initial.

```
(define-type Pos (Refine (v : Int) (<= 0 v)))

(define poss : (Listof Pos) (list 0 1 2 3))
```

`Pos` fixed so that `poss` is rejected.

```
(define-type Pos (Refine (v : Int) (< 0 v)))
```

`poss` fixed so that it is accepted.

```
(define poss : (Listof Pos) (list 1 2 3))
```

### A Term Can Have Many Types

```
(define zero\' : Nat zero)
```

### Example: Natural Numbers

```
(define four : Nat (let ((x 3)) (+ x 1)))
```

### Pre-Conditions

```
(: impossible (All (a) (-> (Refine (v : Any) Bot) a)))
(define (impossible msg) (error msg))
```

### Exercise: Pre-Conditions

Initial.

```
(: safeDiv (-> Int Int Int))
(define (safeDiv x n)
  (match n
    (0 (impossible "divide-by-zero"))
    (_ (quotient x n))))
```

`safeDiv` with type fixed.

```
(: safeDiv (-> Int (Refine (z : Int) (not (= z 0))) Int))
(define (safeDiv x n)
  (match n
    (0 (impossible "divide-by-zero"))
    (_ (quotient x n))))
```

### Exercise: Check That Data

I had to define the helper function `readLn`

```
(: readLn (-> Int))
(define (readLn)
  (let ((__n (read-line)))
    (if (string? __n)
        (let ((_n (string->number __n)))
          (if _n
              (exact-truncate (real-part _n))
              (error "input can't be parsed as a number!")))
        (error "input is empty!"))))
```

The initial `calc`. Doesn't typecheck.

```
(: calc (-> Void))
(define (calc)
  (displayln "Enter numerator")
  (define n (readLn))
  (displayln "Enter denominator")
  (define d (readLn))
  (printf "Result = ~v~n" (safeDiv n d)))
```

Fixed `calc`.

```
(: calc (-> Void))
(define (calc)
  (displayln "Enter numerator")
  (define n (readLn))
  (displayln "Enter denominator")
  (define d (readLn))
  (if (= d 0)
      (error "denominator is zero!")
      (printf "Result = ~v~n" (safeDiv n d))))
```

### Precondition Checked at Call-Site

The following function does not compile, just like its LH counterpart.

```
(: avg (-> (Listof Int) Int))
(define (avg xs)
  (let ((total (apply + xs))
        (n     (length xs)))
    ((safeDiv total) n)))
```

### size returns positive values

```
(: size (All (a) (-> (Listof a) Pos)))
(define (size lst)
  (match lst
    (`()         1)
    (`(,x . ,xs) (let ((n (size xs))) (+ 1 n)))))
```

### Verifying avg

```
(: avg\' (-> (Listof Int) Int))
(define (avg\' xs)
  (let ((total (apply + xs))
        (n     (size xs)))
    (safeDiv total n)))
```

### Unfinished Business

This doesn't compile, and it shouldn't.

```
(: size\' (All (a) (-> (Listof a) Pos)))
(define (size\' l)
  (match l
    (`(,x)                      1)
```

```
    (`(_ . ,xs) (+ 1 (size\' xs)))
    (_          (impossible "size"))))
```

## Data Types

### Example: Lists

Typed Racket has no concept of a *measure* on a data type but it's still possible to reason about the length of a list.

When I first approached this chapter I implemented `List` using TR's `Vector` data type because `vector-length` is among the functions that can be used in the predicate part of `Refine`. Most of the subsequent examples and exercises have been implemented using this inefficient vector implementation. I would like to apologize for this and also offer the small consolation that in this specific project the real work is done by the typechecker.

Upon reviewing the vector-based implementation I remembered that I could use `car` in `Refine`'s predicate. This prompted me to try a more efficient implementation where a `List` is an ordinary Typed Racket list paired with its length. I re-wrote the `List` "interface" - `Emp`, `:::`, `head`, `tail` - using this more practical implementation so that you can take advantage of my missteps and do the exercises with it instead of the vector implementation. The relevant code is listed under the next section "Specifying the Length of a List".

My original, vector-based implementation:

```
(define-type (List a) (Vectorof a))

(: Emp (All (a) (-> (List a))))
(define (Emp) (vector))

(: ::: (All (a) (-> a (List a) (List a))))
(define (::: x xs)
  (let ((v (make-vector (+ 1 (vector-length xs)) x)))
    (vector-copy! v 1 xs)
    v))
```

### Specifying the Length of a List

First, `make-vector` needs to be provided an appropriate type. I got this snippet from an article on the Racket Blog, Refinement Types in Typed Racket. Without it, the compiler doesn't know that `(make-vector n ...)` makes a vector of size $n$.

```
(require typed/racket/unsafe)

(unsafe-require/typed/provide
 typed/racket/base
 [make-vector (All (A) (-> ([n : Natural]
                            [val : A])
                           (Refine [v : (Vectorof A)]
                             (= n (vector-length v))))))])
```

Next, amend the types of the previously-defined `Emp` and `:::`

```
(: Emp (All (a) (-> (Refine (l : (List a)) (= (vector-length l) 0)))))
(define (Emp) (vector))

(: ::: (All (a) (-> ((x : a) (xs : (List a)))
                    (Refine (l : (List a))
                            (= (vector-length l) (+ 1 (vector-length xs)))))))
```

```
(define (::: x xs)
  (let ((v (make-vector (+ 1 (vector-length xs)) x)))
    (vector-copy! v 1 xs)
    v))
```

If you're adamant that you don't want to use vectors you can start with the code below. A "sized list" can be thought of as a pair whose first element is a natural number - the size of the sized list - and whose second element is an homogeneous TR list with that number of elements. I will stick with the vector-based representation ahead.

```
(define-type (List-2 a) (Pairof Natural (_List a)))
(define-type (_List a) (U _Emp (_::: a)))
(struct _Emp ())
(struct (a) _::: ((head : a) (tail : (_List a))))

(: Emp-2 (All (a) (-> (Refine (l : (List-2 a))
                              (and (= (car l) 0) (: (cdr l) _Emp))))))
(define (Emp-2)
  (ann (cons   0 (_Emp))
       (Pairof 0 _Emp)))

(: :::-2
   (All (a) (-> ((x : a) (xs : (List-2 a)))
                (Refine (l : (List-2 a)) (and (= (car l) (+ (car xs) 1))
                                              (: (cdr l) (_::: a)))))))
(define (:::-2 x xs)
  (ann (cons   (+ (car xs) 1)                              (_::: x (cdr xs)))
       (Pairof (Refine (z : Natural) (= z (+ (car xs) 1))) (_::: a))))
```

**Exercise: Partial Functions**

```
(: head (All (a) (-> (Refine (l : (List a)) (> (vector-length l) 0)) a)))
(define (head l) (vector-ref l 0))

(: tail (All (a) (-> ((l : (Refine (xs : (List a)) (> (vector-length xs) 0))))
  (Refine (xs : (List a)) (= (vector-length xs) (- (vector-length l) 1))))))
(define (tail l)
  (let ((v (make-vector (- (vector-length l) 1) (head l))))
    (vector-copy! v 0 l 1 (vector-length l))
    v))
```

One caveat of this list-based implementation is that I can't use two `tail-2` operations in a row. I have to use `~tail-2` or `~head-2` after the first `tail-2`. Technically, if a user sticks to `:::-2`, `Emp-2`, `~head-2` and `~tail-2` for creating and manipulating values of type `(List-2 a)` then `~head-2` and `~tail-2` are just as safe as `head-2` and `tail-2`. `(error "list is empty")` will never be executed.

```
(: head-2 (All (a) (-> (Refine (l : (List-2 a)) (: (cdr l) (_::: a))) a)))
(define (head-2 l) (when (_:::? (cdr l)) (_:::-head (cdr l))))

(: tail-2 (All (a) (-> ((l : (Refine
                              (xs : (List-2 a))
                              (and (> (car xs) 0) (: (cdr xs) (_::: a))))))
                       (Refine (xs : (List-2 a)) (= (car xs) (- (car l) 1)))))))
(define (tail-2 l)
  (define _tail : (_List a) (when (_:::? (cdr l)) (_:::-tail (cdr l))))
```

```
  (ann (cons   (- (car l) 1)                                  _tail)
       (Pairof (Refine (n : Natural) (= n (- (car l) 1))) (_List a))))

(: ~head-2 (All (a) (-> (Refine (l : (List-2 a)) (> (car l) 0)) a)))
(define (~head-2 l)
  (match (cdr l)
    ((_Emp)     (error "list is empty"))
    ((_::: h _)                    h)))

(: ~tail-2 (All (a) (-> ((l : (Refine (xs : (List-2 a)) (> (car xs) 0))))
                        (Refine (xs : (List-2 a)) (= (car xs) (- (car l) 1))))))
(define (~tail-2 l)
  (match (cdr l)
    ((_Emp)      (error "list is empty"))
    ((_::: _ xs)
     (ann (cons   (- (car l) 1)                               xs)
          (Pairof (Refine (n : Natural) (= n (- (car l) 1))) (_List a))))))
```

### Naming Non-Empty Lists

```
(define-type (ListNE a) (Refine (l : (List a)) (> (vector-length l) 0)))
```

### A Useful Partial Function: Fold / Reduce

```
(: foldr (All (a b) (-> (-> a b b) b (List a) b)))
(define (foldr f acc l)
  (if (= (vector-length l) 0)
      acc
      (f (head l) (foldr f acc (tail l)))))
```

### A Useful Partial Function: Fold / Reduce

```
(: foldr1 (All (a) (-> (-> a a a) (ListNE a) a)))
(define (foldr1 f l) (foldr f (head l) (tail l)))
```

### Exercise: average

This is the initial `average\'` which doesn't compile.

```
(: average\' (-> (List Int) Int))
(define (average\' l)
  (let ((total (foldr1 + l))
        (n (vector-length l)))
    (safeDiv total n)))
```

`average\'` with a safe input type that's accepted by the compiler.

```
(: average\' (-> (ListNE Int) Int))
(define (average\' l)
  (let ((total (foldr1 + l))
        (n (vector-length l)))
    (safeDiv total n)))
```

### Example: Year is 12 Months

```
(define-type (Year a) (List a))
```

### Example: Year is 12 Months

In Typed Racket, every number, such as 12, is also the type containing itself.

The closest I can get to defining a type like `ListN a N` in TR is

```
(define-type (ListN a n) (Refine (l : (List a)) (: (vector-length l) n)))
```

I can't get the compiler to see that if `(= (vector-length l) 12)` then it's also true that `(: (vector-length l) 12)`.

This setback is only mildly unfortunate because I can still refine `Year` as

```
(define-type (Year a) (Refine (l : (List a)) (= (vector-length l) 12)))
```

The next definition is also handy.

```
(define-type (|List 0| a)  (Refine (l : (List a)) (= (vector-length l) 0)))
```

Once `Year` is refined, the following definition doesn't typecheck.

```
(define badYear : (Year Int) (::: 1 (ann (Emp) (|List 0| Int))))
```

The next definition, on the other hand, typechecks as I hoped.

```
(define goodYear : (Year String)
  (::: "jan" (::: "feb" (::: "mar" (::: "apr"
  (::: "may" (::: "jun" (::: "jul" (::: "aug"
  (::: "sep" (::: "oct" (::: "nov" (::: "dec"
  (ann (Emp) (|List 0| String))))))))))))))))
```

### Exercise: map

The initial `map` with its friend `tempAverage`.

```
(: map (All (a b) (-> (-> a b) (List a) (List b))))
(define (map f l)
  (if (= (vector-length l) 0)
      (Emp)
      (::: (f (head l)) (map f (tail l)))))

(struct Weather ((temp : Int) (rain : Int)))

(: tempAverage (-> (Year Weather) Int))
(define (tempAverage y)
  (let ((months (map Weather-temp y)))
    (average\' months)))
```

The fixed `map`:

```
(: map (All (a b) (-> ((f  : (-> a b))
                       (l : (List a)))
                      (Refine (xs : (List b)) (= (vector-length l)
                                                 (vector-length xs))))))
(define (map f l)
  (if (= (vector-length l) 0)
```

```
      (Emp)
      (::: (f (head l)) (map f (tail l)))))
```

**Exercise: init**

The initial `init` accompanied by `sanDiegoTemp`.

```
(: init (All (a) (-> (-> Int a) Nat (List a))))
(define (init f n)
  (if (<= n 0)
      (Emp)
      (::: (f n) (init f (- n 1)))))

(define sanDiegoTemp : (Year Int) (init (const 72) 12))
```

`init` with its type remedied:

```
(: init (All (a) (-> ((f : (-> Int a)) (n : Nat))
                     (Refine (l : (List a)) (= (vector-length l) n)))))
(define (init f n) (if (= 0 n) (Emp) (::: (f n) (init f (- n 1)))))
```

## Chapter 4, Case Study: Insertion Sort

After many unsuccessful attempts, I have become convinced that it is not possible to implement in Typed Racket

- an equivalent of the `elems` measure
- an ordered list type such that not only remembers that its head element is not greater than any element in its tail but also remembers that the head element *of its tail* is not greater than any element in the tail of its tail and so on

The best I have been able to do is define a data type that allows you to construct an ordered list but when you destruct it you lose the guarantee that it's ordered.

If I have not managed to reproduce a feature in TR I have omitted its sections from this document.

**Insertion Sort**

```
(: sort (-> (List Int) (List Int)))
(define (sort l)
  (if (= (vector-length l) 0)
      (Emp)
      (insert (head l) (sort (tail l)))))

(: insert (-> Int (List Int) (List Int)))
(define (insert x l)
  (if (= (vector-length l) 0)
      (::: x l)
      (let ((y  (head l))
            (ys (tail l)))
        (if (<= x y)
            (::: x (::: y ys))
            (::: y (insert x ys))))))
```

### Exercise: insert

`sort` and `insert`, where the type of `insert` needs to be fixed.

```
(: sort
  (-> ((l : (List Int)))
    (Refine (m : (List Int)) (= (vector-length m) (vector-length l)))))
(define (sort l)
  (if (= (vector-length l) 0)
      (Emp)
      (insert (head l) (sort (tail l)))))

(: insert (-> Int (List Int) (List Int)))
(define (insert x l)
  (if (= (vector-length l) 0)
      (::: x l)
      (let ((y  (head l))
            (ys (tail l)))
        (if (<= x y)
            (::: x (::: y ys))
            (::: y (insert x ys))))))
```

`insert` with its type fixed:

```
(: insert (-> ((x : Int) (l : (List Int)))
              (Refine (m : (List Int)) (= (vector-length m)
                                          (+ (vector-length l) 1)))))
(define (insert x l)
  (if (= (vector-length l) 0)
      (::: x l)
      (let ((y  (head l))
            (ys (tail l)))
        (if (<= x y)
            (::: x (::: y ys))
            (::: y (insert x ys))))))
```

### Refined Data: Ordered Pairs

```
(define-type OrdPair (Pairof Int Int))
```

### Exercise: Ordered Pairs

`OrdPair` can be refined to legal values only in this way:

```
(define-type OrdPair (Refine (p : (Pairof Int Int)) (< (car p) (cdr p))))

(define okPair : OrdPair '(2 . 4))
```

The typechecker doesn't accept `badPair` as valid.

```
(define badPair : OrdPair '(4 . 2))
```

### Refined Data: CSV Tables

```
(define-type Csv (Pairof (List String) (List (List Int))))
```

```
(define scores : Csv
  (let ()
    (define EmpS : (-> (List String))     Emp)
    (define EmpI : (-> (List Int))        Emp)
    (define EmpL : (-> (List (List Int))) Emp)
    (cons (::: "Id" (::: "Midterm" (::: "Final" (EmpS))))
     (::: (::: 1    (::: 25        (::: 88      (EmpI))))
     (::: (::: 2    (::: 27        (::: 83      (EmpI))))
     (::: (::: 3    (::: 19        (::: 93      (EmpI))))
     (EmpL)))))))
```

### Exercise: Valid CSV Tables

I didn't refine the type `Csv` and instead wrote a function that only constructs values of type `Csv` from "acceptable" pairs of arguments.

```
(: Csv-mk (-> ((hdrs : (List String))
               (vals : (hdrs) (List (Refine (row : (List Int))
                                            (= (vector-length row)
                                               (vector-length hdrs))))))
              Csv))
(define (Csv-mk hdrs vals)
  (cons hdrs (map (lambda ((x : (List Int))) x) vals)))
```

Here is `scores` constructed with `Csv-mk` instead of `cons`.

```
(define-type (|List 3| a) (Refine (l : (List a)) (= (vector-length l) 3)))
(define scores : Csv
  (let ()
    (define EmpS : (-> (|List 0| String))         Emp)
    (define EmpI : (-> (|List 0| Int))            Emp)
    (Csv-mk (::: "Id" (::: "Midterm" (::: "Final" (EmpS))))
     (::: (ann (::: 1 (::: 25 (::: 88 (EmpI)))) (|List 3| Int))
     (::: (ann (::: 2 (::: 27 (::: 83 (EmpI)))) (|List 3| Int))
     (::: (ann (::: 3 (::: 19 (::: 93 (EmpI)))) (|List 3| Int))
     (ann (Emp) (|List 0| (|List 3| Int)))))))))
```

`scores\'` below is not accepted by the typechecker.

```
(define-type (|List 2| a) (Refine (l : (List a)) (= (vector-length l) 2)))
(define scores\' : Csv
  (let ()
    (define EmpS : (-> (|List 0| String))         Emp)
    (define EmpI : (-> (|List 0| Int))            Emp)
    (Csv-mk (::: "Id" (::: "Midterm" (::: "Final" (EmpS))))
     (::: (ann (::: 1 (::: 25 (::: 88 (EmpI)))) (|List 3| Int))
     (::: (ann (::: 2         (::: 83 (EmpI)))  (|List 2| Int))
     (::: (ann (::: 3 (::: 19 (::: 93 (EmpI)))) (|List 3| Int))
     (ann (Emp) (|List 0| (|List 3| Int)))))))))
```

### Lists

It's easier to use TR's existing list type but since I have shadowed the binding of `List` I will use `Listof`.

```
(define-type (OList a) (Listof a))
(define OEmp null)
```

```
(define :<: cons)
```

### Ordered Lists

I don't think it's feasible to refine the type `(OList a)`, instead we replace the existing definition of the constructor `:<:` with a new one. The problem with this kind of list is that it only keeps track of the smallest element in the list. When a new element is added using `:<:` the smallest element changes to the new element *but* when the head element is removed using `oTl` the smallest element remains the old head and does not change to the head of the new list.

If you can find a better way to solve this problem please let me know! Personally I think it's impossible without dependent structs.

```
(: :<: (-> ((oHd : Int) (oTl : (oHd) (OList (Refine (z : Int) (>= z oHd)))))
           (OList (Refine (z : Int) (>= z oHd)))))
(define (:<: oHd oTl) (cons (ann oHd (Refine (z : Int) (>= z oHd))) oTl))

(: oHd (All (a) (-> (OList a) a)))
(define (oHd l) (car l))

(: oTl (All (a) (-> (OList a) (OList a))))
(define (oTl l) (cdr l))
```

### Ordered Lists

The typechecker accepts `okList`.

```
(define okList (:<: 1 (:<: 2 (:<: 3 null))))
```

To demonstrate what I meant about `oTl`:

```
(ann (oTl okList) (OList (Refine (z : Int) (>= z 1))))
```

It rejects `badList`.‘

```
(define badList (:<: 1 (:<: 3 (:<: 2 null))))
```