4 Types in Typed Scheme

Typed Scheme provides a rich variety of types to describe data. This section introduces them.

4.1 Basic Types

The most basic types in Typed Scheme are those for primitive data, such as **`True`** and **`False`** for booleans, **`String`** for strings, and **`Char`** for characters.

```
> '"hello, world"
- : String
"hello, world"
> #\f
- : Char
#\f
> #t
- : True
#t
> #f
- : False
#f
```

Each symbol is given a unique type containing only that symbol. The **`Symbol`** type includes all symbols.

```
> 'foo
- : 'foo
foo
> 'bar
- : 'bar
bar
```

Typed Scheme also provides a rich hierarchy for describing particular kinds of numbers.

```
> 0
- : 0
0
> -7
- : Integer
-7
> 14
- : Exact-Positive-Integer
14
> 3.2
- : Flonum
3.2
> 7.0+2.8i
- : Number
7.0+2.8i
```

Finally, any value is itself a type:

```
> (ann 23 : 23)
- : 23
23
```

4.2 Function Types

We have already seen some examples of function types. Function types are constructed using **`->`**, with the argument types before the arrow and the result type after. Here are some example function types:

```
(Number -> Number)
(String String -> Boolean)
(Char -> (values String Natural))
```

The first type requires a **Number** as input, and produces a **Number**. The second requires two arguments. The third takes one argument, and produces multiple values, of types **String** and **Natural**. Here are example functions for each of these types.

```
> (lambda: ([x : Number]) x)
- : (Number -> Number)
#<procedure>
> (lambda: ([a : String] [b : String]) (equal? a b))
- : (String String -> Boolean)
#<procedure>
> (lambda: ([c : Char]) (values (string c) (char->integer c)))
- : (Char -> (values String Exact-Nonnegative-Integer))
#<procedure>
```

4.3 Union Types

Sometimes a value can be one of several types. To specify this, we can use a union type, written with the type constructor **U**.

```
> (let ([a-number 37])
    (if (even? a-number)
        'yes
        'no))
- : (U 'yes 'no)
no
```

Any number of types can be combined together in a union, and nested unions are flattened.

```
(U Number String Boolean Char)
```

4.4 Recursive Types

Recursive types can refer to themselves. This allows a type to describe an infinite family of data. For example, this is the type of binary trees of numbers.

```
(Rec BT (U Number (Pair BT BT)))
```

The **Rec** type constructor specifies that the type `BT` refers to the whole binary tree type within the body of the **Rec** form.

4.5 Structure Types

Using **define-struct:** introduces new types, distinct from any previous type.

```
(define-struct: point ([x : Real] [y : Real]))
```

Instances of this structure, such as `(make-point 7 12)`, have type `point`.

4.6 Subtyping

In Typed Scheme, all types are placed in a hierarchy, based on what values are included in the type. When an element of a larger type is expected, an element of a smaller type may be provided. The smaller type is called a *subtype* of the larger type. The larger type is called a *supertype*. For example, **Integer** is a subtype of **Real**, since every integer is a real number. Therefore, the following code is acceptable to the type checker:

```
(: f (Real -> Real))
(define (f x) (* x 0.75))

(: x Integer)
(define x -125)
```

```
(f x)
```

All types are subtypes of the **Any** type.

The elements of a union type are individually subtypes of the whole union, so **String** is a subtype of **(U String Number)**. One function type is a subtype of another if they have the same number of arguments, the subtype's arguments are more permissive (is a supertype), and the subtype's result type is less permissive (is a subtype). For example, **(Any -> String)** is a subtype of **(Number -> (U String #f))**.

4.7 Polymorphism

Typed Scheme offers abstraction over types as well as values.

4.7.1 Polymorphic Data Structures

Virtually every Scheme program uses lists and sexpressions. Fortunately, Typed Scheme can handle these as well. A simple list processing program can be written like this:

```
#lang typed/scheme
(: sum-list ((Listof Number) -> Number))
(define (sum-list l)
  (cond [(null? l) 0]
        [else (+ (car l) (sum-list (cdr l)))]))
```

This looks similar to our earlier programs – except for the type of `l`, which looks like a function application. In fact, it's a use of the *type constructor* **Listof**, which takes another type as its input, here **Number**. We can use **Listof** to construct the type of any kind of list we might want.

We can define our own type constructors as well. For example, here is an analog of the `Maybe` type constructor from Haskell:

```
#lang typed/scheme
(define-struct: None ())
(define-struct: (a) Some ([v : a]))

(define-type (Opt a) (U None (Some a)))

(: find (Number (Listof Number) -> (Opt Number)))
(define (find v l)
  (cond [(null? l) (make-None)]
        [(= v (car l)) (make-Some v)]
        [else (find v (cdr l))]))
```

The first **define-struct:** defines `None` to be a structure with no contents.

The second definition

```
(define-struct: (a) Some ([v : a]))
```

creates a parameterized type, `Just`, which is a structure with one element, whose type is that of the type argument to `Just`. Here the type parameters (only one, `a`, in this case) are written before the type name, and can be referred to in the types of the fields.

The type definiton

```
(define-type (Opt a) (U None (Some a)))
```

creates a parameterized type – `Opt` is a potential container for whatever type is supplied.

The `find` function takes a number `v` and list, and produces `(make-Some v)` when the number is found in the list, and `(make-None)` otherwise. Therefore, it produces a `(Opt Number)`, just as the annotation specified.

4.7.2 Polymorphic Functions

Sometimes functions over polymorphic data structures only concern themselves with the form of the structure. For example, one might write a function that takes the length of a list of numbers:

```
#lang typed/scheme
(: list-number-length ((Listof Number) -> Integer))
(define (list-number-length l)
  (if (null? l)
      0
      (add1 (list-number-length (cdr l)))))
```

and also a function that takes the length of a list of strings:

```
#lang typed/scheme
(: list-string-length ((Listof String) -> Integer))
(define (list-string-length l)
  (if (null? l)
      0
      (add1 (list-string-length (cdr l)))))
```

Notice that both of these functions have almost exactly the same definition; the only difference is the name of the function. This is because neither function uses the type of the elements in the definition.

We can abstract over the type of the element as follows:

```
#lang typed/scheme
(: list-length (All (A) ((Listof A) -> Integer)))
(define (list-length l)
  (if (null? l)
      0
      (add1 (list-length (cdr l)))))
```

The new type constructor **All** takes a list of type variables and a body type. The type variables are allowed to appear free in the body of the **All** form.

4.8 Variable-Arity Functions: Programming with Rest Arguments

Typed Scheme can handle some uses of rest arguments.

4.8.1 Uniform Variable-Arity Functions

In Scheme, one can write a function that takes an arbitrary number of arguments as follows:

```
#lang scheme
(define (sum . xs)
  (if (null? xs)
      0
      (+ (car xs) (apply sum (cdr xs)))))

(sum)
(sum 1 2 3 4)
(sum 1 3)
```

The arguments to the function that are in excess to the non-rest arguments are converted to a list which is assigned to the rest parameter. So the examples above evaluate to `0`, `10`, and `4`.

We can define such functions in Typed Scheme as well:

```
#lang typed/scheme
(: sum (Number * -> Number))
(define (sum . xs)
  (if (null? xs)
      0
      (+ (car xs) (apply sum (cdr xs)))))
```

This type can be assigned to the function when each element of the rest parameter is used at the same type.

4.8.2 Non-Uniform Variable-Arity Functions

However, the rest argument may be used as a heterogeneous list. Take this (simplified) definition of the Scheme function `map`:

```
#lang scheme
(define (map f as . bss)
  (if (or (null? as)
          (ormap null? bss))
      null
      (cons (apply f (car as) (map car bss))
            (apply map f (cdr as) (map cdr bss)))))

(map add1 (list 1 2 3 4))
(map cons (list 1 2 3) (list (list 4) (list 5) (list 6)))
(map + (list 1 2 3) (list 2 3 4) (list 3 4 5) (list 4 5 6))
```

Here the different lists that make up the rest argument `bss` can be of different types, but the type of each list in `bss` corresponds to the type of the corresponding argument of `f`. We also know that, in order to avoid arity errors, the length of `bss` must be one less than the arity of `f` (as `as` corresponds to the first argument of `f`).

The example uses of `map` evaluate to `(list 2 3 4 5)`, `(list (list 1 4) (list 2 5) (list 3 6))`, and `(list 10 14 18)`.

In Typed Scheme, we can define `map` as follows:

```
#lang typed/scheme
(: map
   (All (C A B ...)
        ((A B ... B -> C) (Listof A) (Listof B) ... B
         ->
         (Listof C))))
(define (map f as . bss)
  (if (or (null? as)
          (ormap null? bss))
      null
      (cons (apply f (car as) (map car bss))
            (apply map f (cdr as) (map cdr bss)))))
```

Note that the type variable `B` is followed by an ellipsis. This denotes that B is a dotted type variable which corresponds to a list of types, much as a rest argument corresponds to a list of values. When the type of `map` is instantiated at a list of types, then each type `t` which is bound by `B` (notated by the dotted pre-type `t ... B`) is expanded to a number of copies of `t` equal to the length of the sequence assigned to `B`. Then `B` in each copy is replaced with the corresponding type from the sequence.

So the type of `(inst map Integer Boolean String Number)` is

`((Boolean String Number -> Integer) (Listof Boolean) (Listof String) (Listof Number) -> (Listof Integer))`.