Typed Scheme: Scheme with Static Types

Version 4.1.2

Sam Tobin-Hochstadt

October 28, 2008

```
#lang typed-scheme
```

Typed Scheme is a Scheme-like language, with a type system that supports common Scheme programming idioms. Explicit type declarations are required — that is, there is no type inference. The language supports a number of features from previous work on type systems that make it easier to type Scheme programs, as well as a novel idea dubbed *occurrence typing* for case discrimination.

Typed Scheme is also designed to integrate with the rest of your PLT Scheme system. It is possible to convert a single module to Typed Scheme, while leaving the rest of the program unchanged. The typed module is protected from the untyped code base via automatically-synthesized contracts.

Further information on Typed Scheme is available from the homepage.

1 Starting with Typed Scheme

If you already know PLT Scheme, or even some other Scheme, it should be easy to start using Typed Scheme.

1.1 A First Function

The following program defines the Fibonacci function in PLT Scheme:

```
#lang scheme
(define (fib n)
  (cond [(= 0 n) 1]
        [(= 1 n) 1]
        [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

This program defines the same program using Typed Scheme.

```
#lang typed-scheme
(: fib (Number -> Number))
(define (fib n)
  (cond [(= 0 n) 1]
        [(= 1 n) 1]
        [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

There are two differences between these programs:

- **The Language:** `scheme` has been replaced by `typed-scheme`.
- **The Type Annotation:** We have added a type annotation for the `fib` function, using the `:` form.

In general, these are most of the changes that have to be made to a PLT Scheme program to transform it into a Typed Scheme program.

Changes to uses of **`require`** may also be necessary - these are described later.

1.2 Adding more complexity

Other typed binding forms are also available. For example, we could have rewritten our fibonacci program as follows:

```
#lang typed-scheme
(: fib (Number -> Number))
(define (fib n)
```

```
  (let ([base? (or (= 0 n) (= 1 n))])
    (if base?
        1
        (+ (fib (- n 1)) (fib (- n 2))))))
```

This program uses the `let` binding form, but no new type annotations are required. Typed Scheme infers the type of `base?`.

We can also define mutually-recursive functions:

```
#lang typed-scheme
(: my-odd? (Number -> Boolean))
(define (my-odd? n)
  (if (= 0 n) #f
      (my-even? (- n 1))))

(: my-even? (Number -> Boolean))
(define (my-even? n)
  (if (= 0 n) #t
      (my-odd? (- n 1))))

(display (my-even? 12))
```

As expected, this program prints `#t`.

1.3 Defining New Datatypes

If our program requires anything more than atomic data, we must define new datatypes. In Typed Scheme, structures can be defined, similarly to PLT Scheme structures. The following program defines a date structure and a function that formats a date as a string, using PLT Scheme's built-in `format` function.

```
#lang typed-scheme
(define-struct: Date ([day : Number] [month : String] [year : Number]))

(: format-date (Date -> String))
(define (format-date d)
  (format "Today is day ~a of ~a in the year ~a"
          (Date-day d) (Date-month d) (Date-year d)))

(display (format-date (make-Date 28 "November" 2006)))
```

Here we see the new built-in type `String` as well as a definition of the new user-defined type `my-date`. To define `my-date`, we provide all the information usually found in a `define-struct`, but added type annotations to the fields using the `define-struct:` form. Then we can use the functions that this declaration creates, just as we would have with `define-struct`.

1.4 Recursive Datatypes and Unions

Many data structures involve multiple variants. In Typed Scheme, we represent these using *union types*, written `(U t1 t2 ...)`.

```
#lang typed-scheme
(define-type-alias Tree (U leaf node))
(define-struct: leaf ([val : Number]))
(define-struct: node ([left : Tree] [right : Tree]))

(: tree-height (Tree -> Number))
(define (tree-height t)
  (cond [(leaf? t) 1]
        [else (max (tree-height (node-left t))
                   (tree-height (node-right t)))]))

(: tree-sum (Tree -> Number))
(define (tree-sum t)
  (cond [(leaf? t) (leaf-val t)]
        [else (+ (tree-sum (node-left t))
                 (tree-sum (node-right t)))]))
```

In this module, we have defined two new datatypes: `leaf` and `node`. We've also defined the type alias `Tree` to be `(U node leaf)`, which represents a binary tree of numbers. In essence, we are saying that the `tree-height` function accepts a `Tree`, which is either a `node` or a `leaf`, and produces a number.

In order to calculate interesting facts about trees, we have to take them apart and get at their contents. But since accessors such as `node-left` require a `node` as input, not a `Tree`, we have to determine which kind of input we were passed.

For this purpose, we use the predicates that come with each defined structure. For example, the `leaf?` predicate distinguishes `leaf`s from all other Typed Scheme values. Therefore, in the first branch of the `cond` clause in `tree-sum`, we know that `t` is a `leaf`, and therefore we can get its value with the `leaf-val` function.

In the else clauses of both functions, we know that `t` is not a `leaf`, and since the type of `t` was `Tree` by process of elimination we can determine that `t` must be a `node`. Therefore, we can use accessors such as `node-left` and `node-right` with `t` as input.

2 Polymorphism

Typed Scheme offers abstraction over types as well as values.

2.1 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: Nothing ())
(define-struct: (a) Just ([v : a]))

(define-type-alias (Maybe a) (U Nothing (Just a)))

(: find (Number (Listof Number) -> (Maybe Number)))
(define (find v l)
  (cond [(null? l) (make-Nothing)]
        [(= v (car l)) (make-Just v)]
        [else (find v (cdr l))]))
```

The first `define-struct:` defines `Nothing` to be a structure with no contents.

The second definition

```
(define-struct: (a) Just ([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 alias definiton

```
(define-type-alias (Maybe a) (U Nothing (Just a)))
```

creates a parameterized alias — `Maybe` is a potential container for whatever type is supplied.

The `find` function takes a number `v` and list, and produces `(make-Just v)` when the number is found in the list, and `(make-Nothing)` otherwise. Therefore, it produces a `(Maybe Number)`, just as the annotation specified.

2.2 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.

3 Variable-Arity Functions: Programming with Rest Arguments

Typed Scheme can handle some uses of rest arguments.

3.1 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.

3.2 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)).

4 Type Reference

Base Types

These types represent primitive Scheme data.

`Number`

A number

`Integer`

An integer

`Boolean`

Either `#t` or `#f`

`String`

A string

`Keyword`

A literal keyword

`Symbol`

A symbol

`Void`

`#<void>`

`Port`

A port

`Path`

A path

`Char`

A character

`Any`

Any value

The following base types are parametric in their type arguments.

`(Listof t)`

Homogenous lists of *t*

`(Boxof t)`

A box of *t*

`(Vectorof t)`

Homogenous vectors of *t*

`(Option t)`

Either *t* of `#f`

`(Parameter t)`
`(Parameter s t)`

A parameter of *t*. If two type arguments are supplied, the first is the type the parameter accepts, and the second is the type returned.

`(Pair s t)`

is the pair containing *s* as the `car` and *t* as the `cdr`

Type Constructors

```
(dom ... -> rng)
(dom ... rest * -> rng)
(dom ... rest ... bound -> rng)
(dom -> rng : pred)
```

is the type of functions from the (possibly-empty) sequence `dom ...` to the *`rng`* type. The second form specifies a uniform rest argument of type *`rest`*, and the third form specifies a non-uniform rest argument of type *`rest`* with bound *`bound`*. In the third form, the second occurrence of `...` is literal, and *`bound`* must be an identifier denoting a type variable. In the fourth form, there must be only one `dom` and *`pred`* is the type checked by the predicate.

```
(U t ...)
```

is the union of the types *`t`* `...`

```
(case-lambda fun-ty ...)
```

is a function that behaves like all of the *`fun-ty`*s. The *`fun-ty`*s must all be function types constructed with `->`.

```
(t t1 t2 ...)
```

is the instantiation of the parametric type `t` at types *`t1`* *`t2`* `...`

```
(All (v ...) t)
```

is a parameterization of type *`t`*, with type variables *`v`* `...`

```
(List t ...)
```

is the type of the list with one element, in order, for each type provided to the `List` type constructor.

```
(values t ...)
```

is the type of a sequence of multiple values, with types *`t`* `...`. This can only appear as the return type of a function.

```
v
```

where `v` is a number, boolean or string, is the singleton type containing only that value

---

`'sym`

where *`sym`* is a symbol, is the singleton type containing only that symbol

---

`i`

where `i` is an identifier can be a reference to a type name or a type variable

---

`(Rec n t)`

is a recursive type where *`n`* is bound to the recursive type in the body *`t`*

Other types cannot be written by the programmer, but are used internally and may appear in error messages.

`(struct:n (t ...))`

is the type of structures named `n` with field types *`t`*. There may be multiple such types with the same printed representation.

`<n>`

is the printed representation of a reference to the type variable `n`

5 Special Form Reference

Typed Scheme provides a variety of special forms above and beyond those in PLT Scheme. They are used for annotating variables with types, creating new types, and annotating expressions.

5.1 Binding Forms

loop, *f*, *a*, and *v* are names, *t* is a type. *e* is an expression and *body* is a block.

```
(define: v : t e)
(define: (f [v : t] ...) : t . body)
(define: (a ...) (f [v : t] ...) : t . body)
```

These forms define variables, with annotated types. The first form defines *v* with type *t* and value *e*. The second and third forms defines a function *f* with appropriate types. In most cases, use of *:* is preferred to use of `define:`.

```
(let: ([v : t e] ...) . body)
(let: loop : t0 ([v : t e] ...) . body)
```

where *t0* is the type of the result of *loop* (and thus the result of the entire expression).

```
(letrec: ([v : t e] ...) . body)
```

```
(let*: ([v : t e] ...) . body)
```

```
(lambda: ([v : t] ...) . body)
(lambda: ([v : t] ...    v : t)  . body)
```

```
(plambda: (a ...) ([v : t] ...) . body)
(plambda: (a ...) ([v : t] ...    v : t)  . body)
```

```
(case-lambda: [formals body] ...)
```

where *formals* is like the second element of a `lambda:`

```
(pcase-lambda: (a ...) [formals body] ...)
```

where *formals* is like the second element of a `lambda:`.

5.2 Structure Definitions

```
(define-struct: name ([f : t] ...))
(define-struct: (name parent) ([f : t] ...))
(define-struct: (v ...) name ([f : t] ...))
(define-struct: (v ...) (name parent) ([f : t] ...))
```

Defines a structure with the name *name*, where the fields *f* have types *t*. The second and fourth forms define *name* to be a substructure of *parent*. The last two forms define structures that are polymorphic in the type variables *v*.

5.3 Type Aliases

```
(define-type-alias name t)
(define-type-alias (name v ...) t)
```

The first form defines *name* as type, with the same meaning as *t*. The second form is equivalent to `(define-type-alias name (All (v ...) t))`. Type aliases may refer to other type aliases or types defined in the same module, but cycles among type aliases are prohibited.

5.4 Type Annotation and Instantiation

```
(: v t)
```

This declares that *v* has type *t*. The definition of *v* must appear after this declaration. This can be used anywhere a definition form may be used.

`#{v : t}` This declares that the variable `v` has type `t`. This is legal only for binding occurences of *v*.

```
(ann e t)
```

Ensure that `e` has type `t`, or some subtype. The entire expression has type `t`. This is legal only in expression contexts.

`#{e :: t}` This is identical to `(ann e t)`.

```
(inst e t ...)
```

Instantiate the type of `e` with types `t ...`. `e` must have a polymorphic type with the appropriate number of type variables. This is legal only in expression contexts.

`#{e @ t ...}` This is identical to `(inst e t ...)`.

5.5 Require

Here, `m` is a module spec, `pred` is an identifier naming a predicate, and `r` is an optionally-renamed identifier.

```
(require/typed r t m)
(require/typed m [r t] ...)
```

The first form requires `r` from module `m`, giving it type `t`. The second form generalizes this to multiple identifiers.

```
(require/opaque-type t pred m)
```

This defines a new type `t`. `pred`, imported from module `m`, is a predicate for this type. The type is defined as precisely those values to which `pred` produces `#t`. `pred` must have type `(Any -> Boolean)`.

```
(require-typed-struct name ([f : t] ...) m)
```