A Type Notation for Scheme

by Gary T. Leavens, Curtis Clifton, and Brian Dorn

TR #05-18a
August 2005, revised January 2006

Copyright © 2005-6 by Gary T. Leavens, Curtis Clifton, and Brian Dorn

This document is distributed under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1, or (at your option) any later version.

Version $Revision: 1.44 $ of $Date: 2006/01/18 22:42:55 $

Department of Computer Science, Iowa State University
226 Atanasoff Hall
Ames, Iowa 50011-1041, USA

Table of Contents

1 Overview of the Type System 1

1.1 Tool Support Overview 1
1.2 Outline 2

2 Notation for Type Expressions 3

2.1 Basic Types 4
2.1.1 Datum 4
2.1.2 Void 5
2.1.3 Poof 5
2.2 Applied Types 5
2.3 Procedure Types 6
2.3.1 Procedure Types with Fixed Numbers of Arguments 6
2.3.2 Variable Argument Procedure Types 7
2.4 Polymorphic Types 8
2.5 Intersection Types 10
2.6 Variant Record Types 11
2.7 Type Predicate Types 12

3 Additions to Scheme 13

3.1 Expressions Added To Scheme 13
3.1.1 Cases Expressions 13
3.1.2 Has-Type Expressions 14
3.1.3 Test-Type Expressions 14
3.1.4 Trusted Has-Type Expressions 15
3.1.5 Arrow Expressions 16
3.1.6 Forall Expressions 16
3.2 Definitions Added To Scheme 17
3.2.1 Deftype 17
3.2.2 Define-Datatype Definitions 18
3.3 Top-level Forms Added To Scheme 19
3.3.1 Module 20
3.3.2 Provide 21
3.3.3 Require 21
3.3.4 Require-for-syntax 22
3.3.5 Defrep 22

4 Type Checker Commands and Options 25

5 Future Work and Conclusions 26

Appendix A Grammar Summary 27

A.1 Notation for Type Expressions 27

A.2 Additions to Scheme 27

Bibliography 30

1 Overview of the Type System

The Typedscm dialect of Scheme is a variant on the Scheme dialect used in the book *Essentials of Programming Languages* [Friedman-Wand-Haynes01]. Typedscm adds facilities for declaring types in a Scheme-inspired notation. For example, one can write

```
(deftype add1 (-> (number) number))
(define add1 (lambda (n) (+ n 1)))
```

Typedscm allows for both type checking and type inference; that is, type declarations can be omitted in many cases. For example, Typedscm would infer the type of `add1` even if the `deftype` declaration above were not given.

The rest of this section describes the tool support we provide and gives a brief overview of the rest of this document.

1.1 Tool Support Overview

Typedscm also comes with tool support, namely a type checker, which is itself written in Typedscm.

Our type checker for Scheme is invoked either by a button in DrScheme, or by loading the file '`chez/tc-eval.scm`' into Scheme [R5RS] from a library. This can also be done automatically by using a special command at the operating system prompt, such as `bin/mzscheme-typed` or `bin/scheme-typed`, which loads the appropriate files.

When using DrScheme, once you have selected the Typedscm language (from the Language menu) you can use Typedscm in writing Scheme code. All the syntax is implicitly available at this point. (See Section 3.3.1 [Module], page 20, for details on using Typedscm inside modules.)

In DrScheme, pressing the "Type Check" button in DrScheme type checks the file in the Definitions window, and if there are any type errors, these are displayed in a popup window.

If desired, in DrScheme one can also get a type checked read-eval-print loop by first selecting the Typedscm langauge from the Language menu, and then executing the following two forms in the interactions window.

```
(require (lib "tc-type-check-and-eval-loop.scm" "typedscm"))
(type-check-and-eval-loop)
```

If you are using Chez Scheme, then loading the file '`chez/tc-eval.scm`' is the way to start a new read-eval-print loop. (The contents of this file is essentially the two lines above. As mentioned above, various scripts in the '`bin`' directory do this automatically.) The read-eval-print loop reads a Scheme expression from the user's input, type checks it, and if it does type check, proceeds to evaluate it using the standard scheme evaluator. If the expression seems to contain a type error, execution is not attempted, and instead an error message is printed.

Scheme files loaded or required from the read-eval-print loop are also type checked, as are files that are loaded or required from other files being loaded and required, etc. Scheme files may themselves contain type declarations.

A Scheme file with our type annotations can be used in either DrScheme or Chez Scheme. In DrScheme, one simply uses the "TypedScm" language. This

automatically includes Scheme macros and procedures, found in the library file '`drscheme/tc-ignore-types-at-runtime.scm`'. If you are using Chez Scheme you must use the library file '`chez/tc-ignore-types-at-runtime.scm`'; this file is also loaded automatically by the Unix command `bin/scheme-untyped`.

See the '`README.txt`' file in the distribution for detailed installation instructions.

1.2 Outline

The following sections describe the notation for type expressions used in the checker, our additions to Scheme for declaring various kinds of type information, some hints on what type errors may mean, and various configuration parameters and commands recognized by the type checker. See Appendix A [Grammar Summary], page 27, for a summary of the type checker's additions to Scheme's syntax.

2 Notation for Type Expressions

In this section we discuss the notations used in type expressions and their meaning.

A *type* can be thought of as a set of values with certain associated procedures. For example we use `number` as the name of the type of all numbers (both exact and inexact, real and integer) in Scheme. The procedures that work on numbers include `+` and `*`.

Types can also be thought of as abstractions of the type predicates found in Scheme. For example, the type `number` is an abstraction of the built-in Scheme predicate `number?` in the sense that if `(number?` x`)` is true, then x has type `number`. The following table gives an overview of this abstraction, by relating types to type predicates, and by giving some examples of expressions that satisfy the given predicates. (If this is confusing at this point, you may want to come back to it after you have read about the notation below.)

Correspondence between types, predicates, and expressions

Type expression	Type predicate	Example expressions that: have type, satisfy predicate
`boolean`	`boolean?`	`#t, #f`
`char`	`char?`	`#\a, #\Z, #\newline`
`number`	`number?`	`0, 342.54, -3`
`input-port`	`input-port?`	`(current-input-port)`
`output-port`	`output-port?`	`(current-output-port)`
`string`	`string?`	`"a string", ""`
`symbol`	`symbol?`	`'a-symbol, (quote this)`
`datum`	`datum?`	`#t, #\a, 0, "a str", sym`
`poof`	`poof?`	`(error "bad index")`
`void`	`void?`	`(newline)`
`(list-of number)`	`(list-of number?)`	`(list 3 4), '(2 2 7), '()`
`(list-of datum)`	`(list-of datum?)`	`(list 3 #t 3 #\a), '()`
`(list-of char)`	`(list-of char?)`	`(list #\a #\b), '()`
`(vector-of number)`	`(vector-of number?)`	`(vector 3 4), '#(2 2 7), '#()`
`(vector-of char)`	`(vector-of char?)`	`(vector #\a #\b), '#()`
`(pair-of char char)`	`(pair-of char? char?)`	`(cons #\a #\b), '(#\a . #\b)`
`(-> () char)`	`(-> () char?)`	`(lambda () #\a)`
`(-> () void)`	`(-> () void?)`	`newline`
`(-> (char) boolean)`	`(-> (char?) boolean?)`	`(lambda (c) (eqv? #\a c))`
`(-> (number ...) number)`	`(-> (number? ...) number?)`	`+, *`

The non-standard type predicates that are used above are all defined in two files named '`tc-ignore-types-at-runtime.scm`', one of which is found in the type checker's '`drscheme`' directory, and the other of which is found in its '`chez`' directory.

To begin explaining all of this, we start with the grammar for type expressions.

type-exp ::= *monomorphic-type*
| *polymorphic-type*
monomorphic-type ::= *basic-type*
| *applied-type*
| *procedure-type*
| *intersection-type*
| *variant-record-type*
| *type-predicate-type*

Type expressions are pieces of syntax that denote various types. The rest of this section explains each of these pieces of the syntax. See Section 2.4 [Polymorphic Types], page 8, for the syntax of *polymorphic-type*.

2.1 Basic Types

The syntax of a *basic-type* is as follows.

basic-type ::= *identifier*
identifier ::= an <identifier> in the Scheme syntax [R5RS].

A basic type is either a built-in type, such as `number`, a type variable (see Section 2.4 [Polymorphic Types], page 8), or a user-defined type (see Section 3.2.2 [Define-Datatype Definitions], page 18).

Some of the built-in types are the types used for atomic data items in Scheme. These are displayed in the following table.

Built-in Basic Types For Atomic Scheme Data

Name	Example expressions that have that type
`boolean`	`#t, #f`
`char`	`#\a, #\Z`
`number`	`0, 342.54, -3`
`string`	`"a string", ""`
`symbol`	`'a-symbol, (quote this)`
`input-port`	`(current-input-port)`
`output-port`	`(current-output-port)`

The type checker also knows about several other special basic types: `datum`, `poof`, and `void`. These are described below.

2.1.1 Datum

The built-in basic type `datum` is the type of all the values in Scheme. Every Scheme object has this type; that is, `datum` is the supertype of all types. (In the literature on type checking, such a type is sometimes called "top".)

One use of `datum` is in type checking heterogeneous lists. In Scheme the elements of a list do not all have to be of the same type; such a list is *heterogeneous*. A list whose elements are all of the same type is *homogeneous*. The type checker uses types of the form `(list-of` *T*`)` for lists whose elements all have some type *T*. (See Section 2.2 [Applied Types], page 5, for the meaning of `list-of`.) For example, a list of booleans such as `(#t #f)` has the type

`(list-of boolean)`, and a list of numbers such as `(3 4 2)` has the type `(list-of number)`. For a list such as `(#t 3 #f)`, the type checker uses the type `(list-of datum)`.

Although `datum` is the supertype of all types, the type checker does not infer that an expression has type `datum` unless forced to do so [Jenkins-Leavens96]. One reason for this is to give better error messages, since otherwise every expression would have a type `datum`.

2.1.2 Void

The built-in basic type `void`, as in C, C++, or Java, is used to represent the return type of a procedure that returns but doesn't return a useful result. For example, Scheme's procedure `display` has the following type.

```
(-> (datum) void)
```

This means that a call such as `(display 'Scheme)` has no value, and should only be used for its side effects.

See Section 2.3 [Procedure Types], page 6, for an explanation of the `->` syntax.

2.1.3 Poof

The built-in basic type `poof` is used for the return types of procedures that do not return to their caller. An example is the built-in Scheme procedure `error`, whose type is the following.

```
(-> (datum ...) poof)
```

This means that a call such as `(error 'wrong!)` does not return to its caller.

The type checker considers `poof` to be a subtype of all types [Jenkins-Leavens96]. The reason for this is to allow code such as the following to type check.

```
(if (not (zero? denom))
    (/ num denom)                  ; has type: number
    (error "zero denominator!"))   ; has type: poof
```

It is sensible that the above expression has type `number`, because when it has a value at all, that value is of type `number`.

2.2 Applied Types

The syntax of an *applied-type* is as follows. The notation `{` *type-exp* `}+` means one or more repetitions of the nonterminal *type-exp*.

applied-type ::= `(` *identifier* `{` *type-exp* `}+` `)`

Applied types denote the result of applying some *type constructor*, such as `list-of` or `vector-of` to one or more *type parameters*. For example, we write `(list-of number)` for the type of lists of numbers. The lists `(1 2 3)` and `(3.14 2.73)` both have this type. In the type `(list-of number)`, `list-of` is the type constructor, and `number` is the only type parameter. The application of a type constructor to one or more type parameters produces an *instance* of that type constructor.

Some of the built-in type constructors are used for the types of non-atomic data in Scheme. These are displayed in the following table.

Example Instances of Built-in Type Constructors

Instance	Example expressions that have that type
`(list-of number)`	`(list 3 4 2)`
`(list-of boolean)`	`(cons #t (cons #f '()))`
`(vector-of number)`	`(vector 5 4)`
`(pair-of number number)`	`(cons 3 4)`

Type constructors are symbols, and cannot be type variables (see Section 2.4 [Polymorphic Types], page 8).

2.3 Procedure Types

The syntax of procedure types is given below.

procedure-type ::= `(` `->` `(` `{` *type-exp* `}`* `)` *type-exp* `)`
| `(` `->` `(` `{` *type-exp* `}`* *type-exp* `...` `)` *type-exp* `)`

In the above, the notation `{` *type-exp* `}`* means zero or more repetitions of the nonterminal *type-exp*. The symbol `...` used in the second alternative is a terminal symbol in the grammar; it is used in the types of variable argument procedures.

We explain the two different kinds of procedure types below.

2.3.1 Procedure Types with Fixed Numbers of Arguments

A procedure type denotes the type of a Scheme procedure. In the notation, the types of the arguments are enclosed in parentheses, and these are followed by the type that the procedure returns. For example, the Scheme procedure `substring` has the following type.

```
(-> (string number number) string)
```

This means that `substring` is a procedure with 3 arguments, the first of type `string`, the second of type `number`, and the third of type `number`, which returns a `string`.

You can derive such a type from a lambda expression systematically. Take the lambda expression, replace `lambda` by `->`, replace each formal parameter name by its type, and replace the body by its type. For example, the type of the lambda expression

```
(lambda (n m) (+ 2 (+ n m)))
```

is the following.

```
(-> (number number) number)
```

As another example, the type of `sum` in the following definition

```
(define sum
   (lambda (ls)
     (if (null? ls)
         0
        (+ (car ls) (sum (cdr ls))))))
```

is the following.

```
(-> ((list-of number)) number)
```

Here's an example of a curried procedure, `append3-c`,

```
(define append3-c
  (lambda (ls1)
    (lambda (ls2)
      (lambda (ls3)
        (append ls1 (append ls2 ls3))))))
```

which has a the following "curried" procedure type.

```
(forall (T)
   (-> ((list-of T))
       (-> ((list-of T))
           (-> ((list-of T))
               (list-of T)))))
```

The type above for `append3-c` is polymorphic (see Section 2.4 [Polymorphic Types], page 8). The way to read it is to say that, for all types T, the procedure takes a list of elements of type T and returns a procedure; that procedure in turn takes a list of elements of type T and returns a procedure; that procedure in turn takes a list of elements of type T and returns a list of elements of type T.

2.3.2 Variable Argument Procedure Types

A variable argument procedure type denotes the type of a procedure constructed with Scheme's "unrestricted lambda". The notation is just like that of a normal procedure type (see Section 2.3.1 [Procedure Types with Fixed Numbers of Arguments], page 6), except that the last thing in the list of argument types is a type expression followed by `...`. This means that the procedure may take 0 or more actual arguments of the type immediately preceding `...`. For example, consider the type

```
(-> (number ...) number)
```

This is the type of a procedure that takes 0 or more arguments of type number. This is the type of `sum*` in the following.

```
(define sum*
  (lambda args
    (sum args)))
```

It follows that all of the following calls to `sum*` have type `number`.

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

As another example, the lambda expression

```
(lambda (num1 . nums) (sum (cons num1 nums)))
```

has the following type.

```
(-> (number number ...) number)
```

Note that, unlike the type of `sum*`, the above type only permits calls with one or more numbers as arguments, and does not permit calls with no arguments.

2.4 Polymorphic Types

A polymorphic type denotes a family of types, all of which have a similar shape. The shape of the instances in this family are described by the *type-exp* in the following syntax.

polymorphic-type ::= (`forall` *type-formals monomorphic-type*)
type-formals ::= ({ *type-variable* }*)
type-variable ::= *identifier*

Because of the type inference algorithm [Milner78] used in the type checker [Jenkins-Leavens96], the type checker cannot deal with nested polymorphic types. This is why the syntax for *polymorphic-type* only permits a *monomorphic-type* inside the `forall`. See Chapter 2 [Notation for Type Expressions], page 3, for the syntax of *monomorphic-type*.

Indeed, as a predicate transformer, the run-time version of `forall` is the same as `lambda` (see Section 3.1.6 [Forall Expressions], page 16). In type checking, the notion that corresponds to calling this lambda is called instantiation.

To explain what instantiation of a polymorphic type means, consider the following example.

```
(forall (T) (list-of T))
```

If we *instantiate* this type, by supplying a type, such as `number`, for the type variable `T`, we obtain an *instance* of the above type, in this case `(list-of number)`. The following are all instances of the type `(forall (T) (list-of T))`.

```
(list-of number)
(list-of boolean)
(list-of char)
(list-of (vector-of number))
(list-of (vector-of boolean))
(list-of (-> (number) number))
(list-of (-> ((list-of number)) number))
(list-of (list-of number))
(list-of (list-of (list-of boolean)))
```

Each polymorphic type has an infinite number of instances.

Armed with the notion of instantiation, and of the instances of a polymorphic type, we can describe the meaning of a polymorphic type. Using the analogy between types and sets, a polymorphic type denotes an infinite intersection of the sets that its instances denote. Thus, for example, the type `(forall (T) (list-of T))` denotes the infinite intersection of all of its instances, including those listed above. It turns out that the only Scheme value with this type is the empty list, `()`. Another way of saying this is to say that `()` has the type `(list-of` *T*`)` for all types *T*.

Using the analogy between types and predicates, a polymorphic type denotes a predicate transformer. This predicate transformer, when given predicates corresponding to each of the formals, returns a predicate. The returned predicate is the denotation of the corresponding instance of the polymorphic type. For example, the type `(forall (T) (list-of T))` denotes a predicate transformer we write as `(forall (T?) (list-of T?))`, which is equivalent to `(lambda (T?) (list-of T?))`. When applied to a type predicate, such as `number?`, this predicate transformer returns the predicate `(list-of number?)`. Since the predicate `number?` corresponds to the type `number`, this means that the predicate `(list-of`

`number?`) should correspond to the instance (`list-of number`), which it does. This can be illustrated as follows.

```
(forall (T) (list-of T))  -- denotes -->  (forall (T?) (list-of T?))
           !
           !                                          ||
           !
           !                              (lambda (T?) (list-of T?))
           !                                          !
       applied to                                 applied to
        number                                     number?
        yields                                     yields
           !                                          !
           !                                          !
           v                                          v
   (list-of number)    -- denotes -->        (list-of number?)
```

Another way to illustrate this analogy is through the following table.

Types generators, predicate transformers, and expressions

```
Type                  Type                    Example expressions that:
expression          predicate                have type, satisfy predicate
                     transformer             for all values of formals
-------------------------------------------------------------------
(forall (T)         (forall (T?)
  (list-of T))        (list-of T?))       '()

(forall (T)         (forall (T?)
  (-> (T)             (-> (T?)            (lambda (x)
      char))              char?))            #\a)

(forall (S T)       (forall (S? T?)
 (-> (S T)           (-> (S? T?)          (lambda (x y)
     T))                 T?))                y)

(forall (T)         (forall (T?)
 (-> (T T ...)       (-> (T? T? ...)       (lambda args
     T))                 T?))                (car args))

(forall (S T U)     (forall (S? T? U?)
 (all-of             (all-of               cons
  (-> (T              (-> (T?
       (list-of T))        (list-of T?))
      (list-of T))        (list-of T?))
  (-> (S (list-of U)) (-> (S? (list-of U?))
      (list-of            (list-of
       datum))))           datum?))))
```

As illustrated above, polymorphic types are especially useful for describing the types of higher-order functions. For such functions, one often needs to describe procedure types that are not as specific as the ones used in the previous subsection, but which are also not so general as `datum`.

In making instances, it is important to replace type variables consistently throughout. For example, Scheme's built in procedure `map` has the following type.

```
(forall (S T)
  (-> ((-> (S) T) (list-of S)) (list-of T)))
```

This means that when `map` is used, the type checker must find two types S and T such that the first argument to `map` has type `(->` (S) T`)`, and the second has type `(list-of` S`)`; once the specific types for S and T are chosen, consistent replacement dictates that the result of a call will have type `(list-of` T`)`. For example, consider type checking the following call.

```
(map odd? (list 3 4 2 3))
```

Since the type of `odd?` is `(-> (number) boolean)`, the type checker is forced to choose `number` for `S` and `boolean` for `T`; these are the only choices that match the type of `odd?` correctly against the type of `map`'s first argument. Once the type checker has made this choice, however, it is no longer free to use different types for `S` and `T` in checking the rest of the application. So it must check that the second argument in the call has type `(list-of number)`, because it must consistently replace `S` with `number`. Similarly it must give the type of the result as `(list-of boolean)`, because it must consistently replace `T` with `boolean`.

Another way to view this process is that the type checker chooses an instance of `map`'s type, and uses that in type checking the call. The instance it uses is the following.

```
(-> ((-> (number) boolean) (list-of number)) (list-of boolean))
```

What instance would be used in checking the types of the following application?

```
(map (lambda (n) (+ n 1)) '(3 -2 1))
```

Since `datum` is a type expression, it can be used in an instance to replace type variables in a polymorphic type expression. For example, one type for Scheme's `cons` procedure is as follows.

```
(forall (T) (-> (T (list-of T)) (list-of T)))
```

This has the type `(-> (datum (list-of datum)) (list-of datum))` as an instance.

2.5 Intersection Types

The syntax of an *intersection-type* is as follows.

intersection-type ::= `( all-of {` *type-exp* `}+ )`

Here the notation `{` *type-exp* `}+` means one or more repetitions of the nonterminal *type-exp*.

If one thinks of types as sets of values, then an *intersection type* can be thought of as the intersection of several such sets. If one thinks of types as predicates, then an intersection type corresponds to the application of all of the predicates and a conjunction of the results. Therefore, if x has type `(all-of` S T`)`, then x has both type S and type T. For example, since `'()` has both types `(list-of number)` and `(list-of boolean)`, the empty list also has the following type.

```
(all-of (list-of number) (list-of boolean))
```

One use of such intersection types is for procedures that can be used in several ways. For example, Scheme's `cons` procedure can make homogeneous lists, heterogeneous lists, and pairs. Thus a better type for `cons` than the one given above is the following.

```
(forall (S T U V W)
  (all-of (-> (T (list-of T)) (list-of T))
          (-> (S (list-of U)) (list-of datum))
          (-> (V W) (pair-of V W))))
```

The type checker considers the order of types listed in an intersection type when doing type inference [Jenkins-Leavens96]. It will first try to use the first type listed, then the second, and so on. For example, in the expression `(cons 2 '(3))`, the arguments to `cons` are of type `number` and `(list-of number)`, so the type checker uses an instance of the first type in the above intersection type for `cons`, `(-> (T (list-of T)) (list-of T))`, and so the type of the expression is inferred to be `(list-of number)`. However, the type of the expression `(cons 2 '(#t))` is inferred to be `(list-of datum)` since there is no instance of the first type in the intersection type that matches the argument types. (This is because `T` cannot be consistently replaced by both `number` and `boolean`. See Section 2.4 [Polymorphic Types], page 8, for more about consistent replacement.) Finally, the type of the expression `(cons 2 3)` is inferred to be `(pair-of number number)`, since no instance of either of the first two types for `cons` matches the second argument's type.

The type checker never infers that an expression has an intersection type unless it is forced to do so. However, one can use `deftype` (see Section 3.2.1 [Deftype], page 17) to declare such a type. The types of built-in procedures such as `cons` make heavy use of carefully constructed intersection types.

2.6 Variant Record Types

The syntax of an *variant-record-type* is as follows.

variant-record-type ::= (`variant-record` { *variant* }+)
variant ::= (*identifier* { *field-type-binding* }*)
field-type-binding ::= (*identifier type-exp*)

Again, the notation { *variant* }+ means one or more repetitions of the nonterminal *variant*, and the notation { *field-type-binding* }* means zero or more repetitions of the nonterminal *field-type-binding*,

A variant record type is the type of a name defined with `define-datatype` (see Section 3.2.2 [Define-Datatype Definitions], page 18) [Friedman-Wand-Haynes01]. It lists the complete set of variants and their field-type bindings. Each field-type binding associates a field name with the name of that field's type. The order among the field-type bindings in a *variant* is significant, but the order among the *variant*s themselves is not.

For example, given the following `define-datatype` declaration

```
(define-datatype pi-calc-expr pi-calc-expr?
  (parallel-composition (list-of pi-calc-expr?))
  (restriction (name symbol?) (body pi-calc-expr?))
  (input (name symbol?) (formals (list-of symbol?))
         (body pi-calc-expr?))
  (output (name symbol?) (actuals (list-of symbol?)))
  (replication (body pi-calc-expr?))
  (null ))
```

then the name `pi-calc-expr` has the following type:

```
(variant-record
  (parallel-composition (list-of pi-calc-expr))
  (restriction (name symbol) (body pi-calc-expr))
  (input (name symbol) (formals (list-of symbol))
         (body pi-calc-expr))
  (output (name symbol) (actuals (list-of symbol)))
  (replication (body pi-calc-expr))
  (null ))
```

A variant record S is a *subtype* of another variant record type T if for each *variant* of S there is a corresponding *variant* in T with the same *identifier*, the same number of fields, and such that the nth field in the *variant* of S, the field type is a subtype of the field type of T's nth field.

2.7 Type Predicate Types

The syntax of an *type-predicate-type* is as follows.

type-predicate-type ::= `( type-predicate-for` *type-exp* `)`

A type predicate is a procedure that can be applied to any Scheme object, and yields a boolean value; hence a type predicate type such as `(type-predicate-for number)` is a subtype of `(-> (datum) boolean)`, meaning that a type predicate can be used anywhere a procedure of type `(-> (datum) boolean)` is required. However, type predicate types are used by the type checker to infer the type tested for by the Scheme expressions used in `define-datatype` declarations (see Section 3.2.2 [Define-Datatype Definitions], page 18).

The following are some examples of type predicate types and expressions that have those types.

Type expression	Expressions that: have that type
`(type-predicate-for boolean)`	`boolean?`
`(type-predicate-for char)`	`char?`
`(type-predicate-for number)`	`number?`
`(type-predicate-for string)`	`string?`
`(type-predicate-for symbol)`	`symbol?`
`(type-predicate-for datum)`	`datum?`
`(type-predicate-for (list-of number))`	`(list-of number?)`
`(type-predicate-for (list-of char))`	`(list-of char?)`
`(type-predicate-for (vector-of char))`	`(vector-of char?)`

3 Additions to Scheme

This section describes additions that the type checker makes to Scheme's expressions, definitions, top-level forms, and programs.

3.1 Expressions Added To Scheme

The type checker adds the following syntax to Scheme *expressions*. See the formal syntax in the *Revised(5) Report* on Scheme [R5RS] for the syntax of Scheme's standard expression, i.e., for other alternative productions for *expression*.

expression ::= *cases-exp*
| *has-type-exp*
| *test-type-exp*
| *has-type-trusted-exp*
| *arrow-exp*
| *forall-exp*
| a standard Scheme <expression> from [R5RS]

The details of this syntax are explained in the following subsections.

3.1.1 Cases Expressions

The *cases-exp* is part of the `define-datatype` mechanism found in the second edition of *Essentials of Programming Languages* [Friedman-Wand-Haynes01]. The syntax for a *cases-exp*, adapted from [Friedman-Wand-Haynes01] (p. 47), is given below. The notation { *field-name* }* means zero or more repetitions of the nonterminal *field-name*.

cases-exp ::= (`cases` *type-name expression*
{ *cases-clause* }+)
| (`cases` *type-name expression*
{ *cases-clause* }*
(`else` *sequence*))
type-name ::= *identifier*
cases-clause ::= (*variant-name* ({ *field-name* }*) *sequence*)
variant-name ::= *identifier*
field-name ::= *identifier*
sequence ::= 0 or more <command>s followed by an *expression* [R5RS]

For example, suppose we have the following definition (see Section 3.2.2 [Define-Datatype Definitions], page 18).

```
(define-datatype course course?
    (regular (professor string?) (subject string?) (catnum number?))
    (seminar (leader string?) (subject string?)))
```

Then the following procedures, which uses a *cases-exp*, can extract the subject of a course.

```
(define course->subject
   (lambda (c)
     (cases course c
       (regular (professor subject catnum) subject)
       (seminar (leader subject) subject))))
```

3.1.2 Has-Type Expressions

The syntax of a *has-type-exp* is as follows.

has-type-exp ::= (`has-type` *type-exp expression*)

A *has-type-exp* is used to write a type into an expression. The type checker checks that the expression in the body of `has-type` has the type given, and the run-time value of the whole expression is the value of the body. For example, `(has-type number 3)` type checks because `3` has type `number`; the value of `(has-type number 3)` is 3.

A *has-type-exp* is especially useful with Scheme's `let` and `letrec` expressions, because it allows one to declare the type of a procedure in places where a `deftype` (see Section 3.2.1 [Deftype], page 17) is not permitted. An example is the following.

```
(define fact
  (lambda (n)
    (letrec
       ((fact-iter
          (has-type (-> (number number) number)
                    (lambda (n acc)
                       (if (zero? n)
                            acc
                            (fact-iter (- n 1) (* n acc)))))))
       (fact-iter n 1))))
```

Another use of a *has-type-exp* is to force the type checker to use `datum` (see Section 2.1.1 [Datum], page 4) in a type. For example, one might write:

```
(cons (has-type datum 1) '())
```

which will force the type checker to infer the type `(list-of datum)` for this expression, instead of `(list-of number)`.

3.1.3 Test-Type Expressions

The syntax of a *test-type-exp* is as follows.

test-type-exp ::= (`test-type` *expression expression*)

A *test-type-exp* has two subexpressions. The first must be a type predicate; that is, it must have a type `(type-predicate-for` T`)`, where T is some type. This type predicate is used to test the result of the second expression at runtime. If this test passes, then the value of the second expression is returned, otherwise the expression results in an error. Thus, the meaning of a *test-type-exp* is the meaning of the second expression, provided its value passes the type predicate.

Type checking of a *test-type-exp* is done differently than for the `has-type` expression (see Section 3.1.2 [Has-Type Expressions], page 14). The type checker does not perform type checking on the second expression in a *test-type-exp*, but simply accepts that the type of the expression overall is the type, T, that the type predicate tests for. In effect, it waits until runtime to check the type of the second expression. This allows one to suppress error reports when the code is correct but the type checker cannot prove it. For example, the current type checker does not take the flow of control into account; thus when a dynamic test is made to determine the type of a value using a type predicate, the type checker does not use this information. The type checker complains that the following code is not correct,

```
(let ((input (read)))
  (if (number? input)
      (* 3 input)
      (error "please enter a number)))
```

because `input`, it says, is not a number. However, this error message can be avoided by use of `test-type` as follows.

```
(let ((input (test-type number? (read))))
     (* 3 input))
```

Thus the *test-type-exp* expression can be used as an alternative to the *has-type-exp*, to overcome limitations of the type checker.

3.1.4 Trusted Has-Type Expressions

The syntax of a *has-type-trusted-exp* is as follows.

has-type-trusted-exp ::= (`has-type-trusted` *type-exp expression*)

The meaning of a *has-type-trusted-exp* is just like a `has-type` expression (see Section 3.1.2 [Has-Type Expressions], page 14), but it is type checked differently. The type checker does not perform type checking on the expression, but simply accepts the given type. This allows one to suppress error reports when the code is correct but the type checker cannot prove it. For example, the current type checker does not take the flow of control into account; thus when a dynamic test is made to determine the type of a value using a type predicate, the type checker does not use this information. The type checker complains that the following code is not correct,

```
(cond
 ((symbol? exp) (symbol exp))
 ((list? exp) (s-list (parse-s-list exp)))
 (else (error "parse-sym-exp: bad syntax: " exp)))
```

because `exp`, is treated both as a symbol and as a list. However, this error message can be avoided by use of `has-type-trusted` as follows.

```
(has-type-trusted
  datum
  (cond
   ((symbol? exp) (symbol exp))
   ((list? exp) (s-list (parse-s-list exp)))
   (else (error "parse-sym-exp: bad syntax: " exp))))
```

The `has-type-trusted` form should be used sparingly, because one can suppress real errors with it. For example, the use of `has-type-trusted` in the following suppresses a real, but subtle type error, which will now occur at run-time when the procedure is called.

```
(define add-to-list
   (lambda (num ls)
      (if (null? ls)
          '()
          (cons (+ num (car ls))
                (has-type-trusted
                  (list-of number)
                  (add-to-list (cdr ls))))))) ; wrong!
```

(In the example above, the recursive call on the last line doesn't pass the right number of arguments, since it leaves out the argument `num`.)

When possible, use the *test-type-exp* (see Section 3.1.3 [Test-Type Expressions], page 14), which also can be used to avoid type checking, but will do type checking at run-time.

3.1.5 Arrow Expressions

The syntax of an *arrow-exp* is as follows.

arrow-exp ::= `( -> ( { expression }* )` *expression* `)`
| `( -> ( { expression }*` *expression* `. . . )` *expression* `)`

An *arrow-exp* is used to construct a type predicate that corresponds to a procedure type (see Section 2.3 [Procedure Types], page 6). The meaning of an `<arrow-exp>` is the type predicate `procedure?`, except that the type checker can use the type predicates to construct more specific procedure types. For example, the following expression

```
(-> (boolean? number?) boolean?)
```

has type

```
(type-predicate-for (-> (boolean number) boolean))
```

That is, the expression corresponds to the type `(-> (boolean number) boolean)` (see Section 2.7 [Type Predicate Types], page 12). This is more specific than the type `(-> (datum ...) datum)` which the type checker thinks corresponds to the type predicate `procedure?`. (Note that the type of `procedure?` itself is not the issue, the issue is what type corresponds to an object that satisfies the predicate `procedure?`. See Chapter 2 [Notation for Type Expressions], page 3, for a table of such correspondences.)

Since an *arrow-exp* is a type predicate, such an expression can be used as one of the subexpressions of an `arrow-exp`. For example, the following expression

```
(-> ((-> (number? number?) number?) (list-of number?)) number?)
```

corresponds to the following type.

```
(-> ((-> (number number) number) (list-of number)) number)
```

3.1.6 Forall Expressions

The syntax of a *forall-exp* is as follows.

forall-exp ::= `( forall` *formals expression* `)`
formals ::= `<formals>` in the Scheme syntax [R5RS]

A *forall-exp* is used instead of a `lambda` to write type predicates transformers that correspond to polymorphic types (see Section 2.4 [Polymorphic Types], page 8). A *type predicate transformer* is a procedure that takes type predicates as arguments and returns a type predicate as its result.

Each standard formal T in the list of formals has type `(type-predicate-for` T`)`. The *expression* that forms the body of the `forall` expression must have a type that matches `(type-predicate-for` S`)`, for some type S. (Usually, S will depend on T.)

For example, we could write the type predicate transformer `double-list-of` as follows.

```
(define double-list-of
  (forall (T?)
    (list-of (list-of T?))))
```

From this, the type checker will deduce that the type of `list-of` is the following.

```
(forall (T) (-> ((type-predicate-for T))
                (type-predicate-for (list-of (list-of T)))))
```

The `forall` that appears in the type also causes the overall type to be polymorphic, using a `forall` in the type (see Section 2.4 [Polymorphic Types], page 8). This explains the name of the expression.

In execution, a *forall-exp* of the form `(forall` *formals exp*`)` is equivalent to `(lambda` *formals exp*`)`. The reason to not use the equivalent lambda is to tell the the type checker that one is defining a type predicate transformer for a polymorphic type, as described above.

Note, however, that a type predicate transformer is not itself a type predicate; it is a procedure that returns a type predicate when given type predicates as arguments. Hence, one cannot, for example, use a *forall-exp* as a type predicate in a `define-datatype` declaration (see Section 3.2.2 [Define-Datatype Definitions], page 18). For example, the following will cause a run-time error when used:

```
(define-datatype poly-set poly-set?
   (empty)
   (comprehension
     (predicate (forall (T?) (-> (T?) boolean?))))  ; wrong!
```

The problem will surface when trying to build an object of this type using the `comprehension` constructor. For example, the following will cause a run-time error.

```
(cases poly-set (comprehension null?)
  (empty #f)
  (comprehension (predicate) (predicate '())))
```

One way to make this example work is to use the following instead.

```
(define-datatype poly-set poly-set?
   (empty)
   (comprehension
     (predicate (-> (datum?) boolean?))))
```

3.2 Definitions Added To Scheme

The type checker adds the following kinds of definitions to Scheme. These are explained in the subsections below.

definition ::= *deftype*
| *define-datatype*
| a standard Scheme <definition> from [R5RS]

These are explained below.

3.2.1 Deftype

The syntax of a *deftype* is given below. See Chapter 2 [Notation for Type Expressions], page 3, for the syntax of *type-exp*.

deftype ::= (`deftype` *name type-exp*)
name ::= *identifier*

A *deftype* is used to define the type of a name. It can be used anywhere a Scheme *definition* can be used. The type checker will issue an error message if the type inferred for the name does not match the type given.

For example, one might write the following to define the type of the procedure `remove-first`.

```
(deftype remove-first (forall (T) (-> (T (list-of T)) (list-of T))))
```

One can also use a deftype to define the type of a variable that is not a procedure. For example, the following declares that `e` has type `number`.

```
(deftype e number)
```

3.2.2 Define-Datatype Definitions

The *define-datatype* definition form is found in the second edition of *Essentials of Programming Languages* [Friedman-Wand-Haynes01]. The syntax for a *define-datatype*, adapted from page 44 is given below. The notation { *X* }* means zero or more repetitions of *X*, and { *X* }+ means one or more repetitions of *X*. Recall that a *type-name*, *variant-name*, and *field-name* are each Scheme symbols (see Section 3.1.1 [Cases Expressions], page 13).

define-datatype ::= (`define-datatype` *type-name type-predicate-name*
{ (*variant-name* { *field-pred-binding* }*) }+)
field-pred-binding ::= (*field-name predicate-exp*)
type-predicate-name ::= *identifier*
predicate-exp ::= *expression*

The type checker can only deal with *predicate-exp*s that have a type of the form `(type-predicate-for` *S*`)` for some type *S*. See Section 2.7 [Type Predicate Types], page 12, for an explanation of these types. All of the standard type predicates, such as `number?`, `symbol?`, and several that we supply with the type checker, such as `datum?`, have such types. Another way to produce such a type is to apply a type predicate transformer, such as `list-of`, `vector-of`, `pair-of`, or `->` to known type predicates. Thus the following are all *predicate-exp*s that can be understood by the type checker; they are listed with their corresponding types.

Built-in type predicates and corresponding types

Type predicate	Corresponding type expression
`boolean?`	`boolean`
`number?`	`number`
`datum?`	`datum`
`(list-of string?)`	`(list-of string)`
`(list-of (list-of string?))`	`(list-of (list-of string))`
`(vector-of number?)`	`(vector-of number)`
`(pair-of char? char?)`	`(pair-of char char)`
`(-> (char?) char?)`	`(-> (char) char)`
`(-> (char? ...) char?)`	`(-> (char ...) char)`

However, an expression such as `(if #t number? boolean?)` does not have such a type. The type checker rejects such predicate expressions, because it does not know how to extract

the corresponding type information from them. When this happens, one can fall back to using a type predicate such as `datum?`.

A *define-datatype* declaration declares a variant record type, a new known type predicate, and the types of the procedures for constructing the variants. It also provides information needed to type check a *cases-exp* (see Section 3.1.1 [Cases Expressions], page 13) that is used with that type.

For example, the following definition

```
(define-datatype person person?
   (student (name string?) (major string?))
   (professor (name string?) (office number?)))
```

has the effect of the following `deftype` definitions.

```
(deftype person? (type-predicate-for person))
(deftype student (-> (string string) person))
(deftype professor (-> (string number) person))
(deftype person
   (variant-record
     (student (name string) (major string))
     (professor (name string) (office number))))
```

For example, the following expressions are both of type `person`

```
(student "Alyssa P. Hacker" "Computing")
(professor "Daniel P. Friedman" 3021)
```

In addition, the following expressions will both return `#t`.

```
(person? (student "Mitchell Wand" "Computer Science"))
(person? (professor "Christopher T. Haynes" 3033))
```

Furthermore, the following expression will return `"Hal Abelson"` (see Section 3.1.1 [Cases Expressions], page 13).

```
(let ((mit-prof (professor "Hal Abelson" 507)))
  (cases person mit-prof
     (professor (name salary) name))
     (student (name major) name))
```

Finally, in this example `person` becomes a known type, and `person?` a known type predicate, which can thus be used in making other simple type predicates.

3.3 Top-level Forms Added To Scheme

The type checker adds the following top-level forms to Scheme. See the *Revised(5) Report* on Scheme [R5RS] for the syntax of Scheme's *command* and *syntax-definition*. See Section 3.2 [Definitions Added To Scheme], page 17, for the syntax of *definition*. The other forms of *command-or-definition* are explained in the subsections below.

command-or-definition ::= *module*
| *require*
| *command*
| *definition*
| *syntax-definition*
| (`begin` { *command-or-definition* }+)

3.3.1 Module

The syntax of a *module* that follows is based on the modules in MzScheme (see Section 5 of the *PLT MzScheme: Language Manual* [Flatt05]). It is much more restrictive, in that only one *provide* form is allowed, and it must come right after the *initial-required-module-name*, and there can only at most one *require* form, which must follow some optional *deftype*s and at most one *require-for-syntax* as shown below.

```
module ::= ( module identifier
                    initial-required-module-name
                    provide
                    { deftype }*
                    require-for-syntax-opt
                    require-opt
                    defrep-opt
                    { definition }* )
initial-required-module-name ::= module-name
module-name ::= identifier
        | unix-relative-path-string
        | ( file path-string )
        | ( lib filename-string { collection-string }* )
unix-relative-path-string ::= string
path-string ::= string
filename-string ::= string
collection-string ::= string
defrep-opt ::=   | defrep
require-for-sytnax-opt ::= | require-for-syntax
require-opt ::=  | require
```

Another difference from MzScheme, is that the above syntax does not allow Scheme *commands* at the top-level of a module body. This allows translation into other Scheme dialects, like Chez Scheme, that do not allow commands in the body of a module.

The meaning of a module is as in MzScheme [Flatt05]. In essence a module defines a scope for names (and syntax), and hides all names (and syntax) that it does not export explicitly. Names can be exported using the *provide* syntax (see Section 3.3.2 [Provide], page 21). Typically one would write *deftype*s for each of these names (see Section 3.2.1 [Deftype], page 17) following the *provide*.

One uses a module by making use of the *require* special form (see Section 3.3.3 [Require], page 21). If a file contains a module, just loading it, with `load`, merely brings that module as a namespace into Scheme, and does not open it up so that its exports can be used. To open up a module so that its exports can be used, you must use the *require* special form (see Section 3.3.3 [Require], page 21).

A module may use *defrep* to declare the representation for one or more abstract datatypes (see Section 3.3.5 [Defrep], page 22).

The *initial-required-module-name* must be `(lib "typedscm.ss" "typedscm")` (or, in theory, `typedscm` if that is already loaded), but could name some other language dialect. Our type checker currently ignores this name, effectively assuming that it is `typedscm`, however, the MzScheme runtime interprets this, and so it is important that your modules

have an *initial-required-module-name* of the form `(lib "typedscm.ss" "typedscm")`. That is, each module written in the Typedscm dialect should start as follows.

```
(module my-module (lib "typedscm.ss" "typedscm")
   ;; ...
)
```

3.3.2 Provide

The syntax of *provide* is as follows.

provide ::= `(` `provide` `{` *identifier* `}*` `)`

A *provide* form allows one to declare which names declared in a module are visible to client code when the module is required (see Section 3.3.3 [Require], page 21). A *provide* form can only appear in a *module* declaration (see Section 3.3.1 [Module], page 20).

Each module must contain a single `provide` form, which lists the names (and syntax) exported to clients that require the module. All other names (and syntax) definitions in the file are hidden in the module and cannot be seen by clients that require the module. This allows information hiding, both about procedures and data structures.

For example, the following defines a module `fib-mod` that exports the procedure `fib`, but hides the helping procedure `fib-iter`.

```
(module fib-mod (lib "typedscm.ss" "typedscm")

  (provide fib)

  (deftype fib (-> (number) number))
  (define fib
    (lambda (n)
      (if (zero? n)
          0
          (fib-iter n 0 1))))

  (deftype fib-iter (-> (number number number) number))
  (define fib-iter
    (lambda (n prev last)
      (if (< n 2)
          last
          (fib-iter (- n 1) last (+ prev last)))))
  )
```

3.3.3 Require

The syntax of *require* is as follows. See Section 3.3.1 [Module], page 20, for the syntax of *module-name*.

require ::= `(` `require` `{` *require-spec* `}+` `)`
require-spec ::= *module-name*

A *require* form allows one to import the names (and syntax) defined by one or more modules into a scope. A *require* form can appear either at the top-level in a program (see

Section 3.3 [Top-level Forms Added To Scheme], page 19) or in a module (see Section 3.3.1 [Module], page 20).

A module can contain at most one `require` form, which lists the modules to be imported. At the point of the `require`, the names (and syntax) defined by each *module-id* are made available in the scope. For details on the semantics, see Section 5 of the *PLT MzScheme: Language Manual* [Flatt05].

For example, the following defines a module `fib2-mod` that imports the `fib-mod` defined above (see Section 3.3.2 [Provide], page 21), assuming that `fib-mod` is contained in a collection named `lib342` and a file named '`fib-mod.scm`'.

```
(module fib2-mod (lib "typedscm.ss" "typedscm")

  (provide fib2 fib)
  (require (lib "fib-mod.scm" "lib342"))

  (deftype fib (-> (number) number))
  (define fib2
    (lambda (n)
      (fib (fib n))))
  )
```

3.3.4 Require-for-syntax

The syntax of *require-for-syntax* is as follows. See Section 3.3.3 [Require], page 21, for the syntax of *require-spec*.

require-for-syntax ::= (`require-for-syntax` { *require-spec* }+)

A *require-for-syntax* form allows one to import names (and syntax) defined by one or more modules into a module's transformer environment. A *require-for-syntax* form can only appear in a *module* declaration (see Section 3.3.1 [Module], page 20).

A module can contain at most one `require-for-syntax` form, which lists the modules to be imported. Bindings imported as a result of `require-for-syntax` are made available during expansion time in MzScheme. For details on the semantics, see Section 12 of the *PLT MzScheme: Language Manual* [Flatt05].

The type checker essentially ignores the *required-for-syntax* form, as bindings imported by it do not impact the type environment of the module. This syntax is allowed primarily so that `define-syntax` within a module can function properly in the MzScheme environment. Most users can safely ignore this feature.

3.3.5 Defrep

The syntax of a *defrep* is as follows. See Section 2.4 [Polymorphic Types], page 8, for the syntax of *type-formals*.

defrep ::= (`defrep` { *abs-conc-pair* }*)
abs-conc-pair ::= (*abstract-type-exp representation-type-exp*)
| (`forall` *type-formals abstract-type-exp representation-type-exp*)
abstract-type-exp ::= *type-exp*
representation-type-exp ::= *type-exp*

A *defrep* form is used to declare the correspondence between abstract data types (ADT) and their concrete representation types [Liskov-Guttag86] [Jenkins-Leavens96]. A *defrep* form can only appear immediately after the *provide* in a *module* definition (see Section 3.3.1 [Module], page 20).

The *abstract-type-exp*s in a `defrep` form should be a basic type or an applied type. If a basic type is used, it cannot be a type variable.

For example, the `defrep` form in the following example, says that the ADT `ratl` is represented by the type `(list-of number)`. The `deftype` declarations present the client's view of the ADT's operations.

```
(module ratl (lib "typedscm.ss" "typedscm")

  (provide make-ratl numr denr)

  (deftype make-ratl (-> (number number) ratl))
  (deftype numr (-> (ratl) number))
  (deftype denr (-> (ratl) number))

  (defrep (ratl (list-of number)))

  ;; the following is from p. 91 of [Springer-Friedman89]

  (define make-ratl
    (lambda (int1 int2)
      (if (zero? int2)
          (error "make-ratl: The denominator cannot be zero.")
          (list int1 int2))))

  (define numr
    (lambda (rtl)
      (car rtl)))

  (define denr
    (lambda (rtl)
      (cadr rtl)))
)
```

Without the `defrep` form, the type checker would infer that the type of the procedure `make-ratl` is as follows,

```
(-> (number number) (list-of number))
```

which would not match the type declared with the `deftype` in the above example.

```
(-> (number number) ratl)
```

However, with the `defrep` form, the type checker matches the inferred type of the procedure against a concrete version of the declared type. This concrete version is formed by substituting the representation type for each occurrence of the abstract type. Thus the concrete version of the declared type `(-> (number number) ratl)` is `(-> (number number) (list-of number))`, and so the inferred type and this concrete version of the declared type match.

The second form of `defrep`, with a `forall` and *type-formals* is used to relate polymorphic types. For example, one might write the following to say that a set of some type, `T` is represented by a list of `T` objects.

```
(defrep (forall (T) (set-of T) (list-of T)))
```

When using the *defrep* form, one must also provide suitable `deftype` declarations (see Section 3.2.1 [Deftype], page 17) for the operations of the ADT, or the `defrep` form will have no effect.

In summary, the type checker uses a `defrep` form to check the concrete types of an ADT implementation against the client-visible types declared using `deftype`.

4 Type Checker Commands and Options

The following commands are helpful to regular users of the type checker.

command	effect
(type-check-help)	displays helpful information
(type-check-exit)	leave the typed interpreter loop
(type-check-and-eval-loop)	starts a typed interpreter loop
(type-check-reset-env!)	forget all names defined so far
(type-check-eval-even-if-error!)	evaluate even if type errors
(type-check-dont-eval-if-error!)	don't eval if type errors

If you have exited the type checker's read-eval-print loop, you can use

```
(type-check-and-eval-loop)
```

to restart it from Scheme.

The following commands are mainly useful for developing the type checker itself.

command	effect
(type-tracing-set! N)	set tracing detail level to N
(type-tracing-off!)	set tracing detail level to 0
(type-tracing-level)	returns the numeric level of tracing detail

5 Future Work and Conclusions

The `define-datatype` facility doesn't support polymorphic variant record types. It seems necessary to have a separate declaration that would allow one to declare polymorphic variant record types, which produces a type predicate transformer instead of a type predicate.

One can also imagine trying to use more sophisticated type inference algorithms that can deal with some uses of nested polymorphic types.

Acknowledgments

Thanks to Steven Jenkins for work on an earlier version of the type checker and for discussions about these ideas. A more extensive (but dated) discussion of the type checker is found in [Jenkins-Leavens96].

The work of Leavens was supported in part by by the US NSF under grants CCR-9803843, CCR-0097907, and CCR-0113181. Some of the work on this document was done while Leavens was a visiting professor at the University of Iowa. Thanks to the students at the University of Iowa in 22C:54, section 2, during Fall 2000. Thanks to the students at Iowa State University in Com S 342 in several semesters.

Appendix A Grammar Summary

The following is a summary of the context-free grammar for the type checker. In the grammar, the notation `{` *X* `}`* means zero or more repetitions of the nonterminal *X*. The notation `{` *X* `}+` means one or more repetitions of the nonterminal *X*.

A.1 Notation for Type Expressions

type-exp ::= *monomorphic-type*
| *polymorphic-type*
monomorphic-type ::= *basic-type*
| *applied-type*
| *procedure-type*
| *intersection-type*
| *variant-record-type*
| *type-predicate-type*
basic-type ::= *identifier*
identifier ::= an <identifier> in the Scheme syntax [R5RS].
applied-type ::= `(` *identifier* `{` *type-exp* `}+` `)`
procedure-type ::= `(` `->` `(` `{` *type-exp* `}*` `)` *type-exp* `)`
| `(` `->` `(` `{` *type-exp* `}*` *type-exp* `...` `)` *type-exp* `)`
polymorphic-type ::= `(` `forall` *type-formals* *monomorphic-type* `)`
type-formals ::= `(` `{` *type-variable* `}*` `)`
type-variable ::= *identifier*
intersection-type ::= `(` `all-of` `{` *type-exp* `}+` `)`
variant-record-type ::= `(` `variant-record` `{` *variant* `}+` `)`
variant ::= `(` *identifier* `{` *field-type-binding* `}*` `)`
field-type-binding ::= `(` *identifier* *type-exp* `)`
type-predicate-type ::= `(` `type-predicate-for` *type-exp* `)`

A.2 Additions to Scheme

expression ::= *cases-exp*
| *has-type-exp*
| *test-type-exp*
| *has-type-trusted-exp*
| *arrow-exp*
| *forall-exp*
| a standard Scheme <expression> from [R5RS]
cases-exp ::= `(` `cases` *type-name* *expression*
`{` *cases-clause* `}+` `)`
| `(` `cases` *type-name* *expression*
type-name ::= *identifier*
cases-clause ::= `(` *variant-name* `(` `{` *field-name* `}*` `)` *sequence* `)`
variant-name ::= *identifier*
field-name ::= *identifier*
sequence ::= 0 or more <command>s followed by an *expression* [R5RS]

```
has-type-exp ::= ( has-type type-exp expression )
test-type-exp ::= ( test-type expression expression )
has-type-trusted-exp ::= ( has-type-trusted type-exp expression )
arrow-exp ::= ( -> ( { expression }* ) expression )
        | ( -> ( { expression }* expression ... ) expression )
forall-exp ::= ( forall formals expression )
formals ::= <formals> in the Scheme syntax [R5RS]
definition ::= deftype
        | define-datatype
        | a standard Scheme <definition> from [R5RS]
deftype ::= ( deftype name type-exp )
name ::= identifier
define-datatype ::= ( define-datatype type-name type-predicate-name
                    { ( variant-name { field-pred-binding }* ) }+ )
field-pred-binding ::= ( field-name predicate-exp )
type-predicate-name ::= identifier
predicate-exp ::= expression
command-or-definition ::= module
        | require
        | command
        | definition
        | syntax-definition
        | ( begin { command-or-definition }+ )
module ::= ( module identifier
                    initial-required-module-name
                    provide
                    { deftype }*
                    require-for-syntax-opt
                    require-opt
                    defrep-opt
                    { definition }* )
initial-required-module-name ::= module-name
module-name ::= identifier
        | unix-relative-path-string
        | ( file path-string )
        | ( lib filename-string { collection-string }* )
unix-relative-path-string ::= string
path-string ::= string
filename-string ::= string
collection-string ::= string
defrep-opt ::=   | defrep
require-for-sytnax-opt ::= | require-for-syntax
require-opt ::=  | require
provide ::= ( provide { identifier }* )
require ::= ( require { require-spec }+ )
require-spec ::= module-name
require-for-syntax ::= ( require-for-syntax { require-spec }+ )
defrep ::= ( defrep { abs-conc-pair  }* )
abs-conc-pair ::= ( abstract-type-exp representation-type-exp )
```

| (`forall` *type-formals abstract-type-exp representation-type-exp*)
abstract-type-exp ::= *type-exp*
representation-type-exp ::= *type-exp*

Bibliography

[Flatt05] Matthew Flatt. *PLT MzScheme: Language Manual*, version 300, December 2005. Also available from '`http://download.plt-scheme.org/doc/300/html/mzscheme/index.htm`'.

[Friedman-Wand-Haynes01] Daniel P. Friedman and Mitchell Wand and Christopher T. Haynes. *Essentials of Programming Languages*, Second Edition, McGraw-Hill, 2001.

[Jenkins-Leavens96] Steven Jenkins and Gary T. Leavens. Polymorphic Type-Checking in Scheme. *Computer Languages*, 22(4):215-223, 1996. Also available from '`ftp://ftp.cs.iastate.edu/pub/techreports/TR95-21/TR.ps.Z`'.

[Liskov-Guttag86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*, McGraw-Hill, 1986.

[Milner78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17(3):348-375, December 1978.

[R5RS] Richard Kelsey, William Clinger, and Jonathan Rees (editors). *Revised(5) Report on the Algorithmic Language Scheme*. February 1998. Available from '`http://www.schemers.org/Documents/Standards/`'.

[Springer-Friedman89] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*, McGraw-Hill, N.Y., 1989.