define-typed: efficient typechecks for Guile Scheme

If you want to add typechecks to Guile Scheme, you can follow the format by sph-sc, a Scheme to C compiler. It declares types after the function definition like this:

```
(define (hello typed-world) (string? string?)
  typed-world)
```

That's simple enough that a plain, hygienic `syntax-rule` can support it.

Contents

1 Usage

Example usage:

```
(define-typed (hello typed-world) (string? string?)
  typed-world)
(hello "typed")
;; => "typed"
(hello 1337)
;; => type error ~a ~a #<procedure string? (_)> 1337
(define-typed (hello typed-world) (string? string?)
  "typed" ;; docstring
  #((props)) ;; more properties
  1337) ;; wrong return type
(procedure-documentation hello)
;; => "typed"
(procedure-properties hello)
```

```
;; =>((argument-types #<procedure string? (_)>)
;;    (return-type . #<procedure string? (_)>)
;;    (name . hello) (documentation . "typed") (props))
(hello "typed")
;; type error: return value ~a does not match ~a (1337) #<procedure string? (_)>
```

Optional and required keyword arguments:

```
(define-typed* (hello #:key typed-world) (string? #:key string?)
  "typed" #((props)) typed-world)
(hello #:typed-world "foo")
;; => "foo"
;; unused keyword arguments are always boolean #f as input
(hello)
;; => type error ~a ~a #<procedure string? (_)> #f
;; typing optional keyword arguments
(define (optional-string? x) (or (not x) (string? x)))
(define-typed* (hello #:key typed-world) (string? #:key optional-string?)
  (or typed-world "world"))
(hello)
;; => "world"
(hello #:typed-world "typed")
;; => "typed"
(hello #:typed-world #t)
;; => type error ~a ~a #<procedure optional-string? (x)> #t
;; optional arguments
(define-typed* (hello #:optional typed-world) (string? #:optional optional-string?)
  (or typed-world "world"))
(hello)
;; => "world"
(hello "typed")
;; => "typed"
(hello #t)
;; => type error ~a ~a #<procedure optional-string? (x)> #t
```

Multiple return values:

```
;; fixed return values
(define-typed
  (multiple-values/fixed num)
  ((number? number?) number?)
  (values (* 2 (abs num)) num))
(multiple-values/fixed -3)
;; => 6
;; => -3
;; check return values via procedure
```

```
(define-inlinable (all-numbers? args)
  (not (member #f (map number? args))))
(define-typed
  (multiple-values/proc num)
  ((all-numbers?) number?)
  (values (* 2 (abs num)) num))
(multiple-values/proc -3)
;; => 6
;; => -3
;; check return values via lambda
(define-typed
  (multiple-values/lambda num)
  (((λ(vals) (apply > vals))) number?)
  (values (* 2 (abs num)) num))
(multiple-values/lambda -3)
;; => 6
;; => -3
```

There are four different ways to check the return types:

```
;; check a single return value
(define-typed (magnitude x y) (float? float? float?) ...)

;; check all return values via procedure that receives them as list
(define-typed (magnitude x y) ((all-float?) float? float?) ...)

;; require a fixed number of return values (2)
(define-typed (magnitude x y) ((float? float?) float? float?) ...)

;; do not check return value(-s): #f as return type skips the check
(define-typed (magnitude x y) (#f float? float?) ...)
```

Checking multiple return values has a negative impact on performance in the current implementation. Checking a single value or skipping the check does not have a significant impact.

2 Implementation

For performance reasons, the following defines `define-typed` and `define-typed*`, where `define-typed*` supports `#:keyword` arguments.

Big thanks to David Thompson and his article Optimizing Guile Scheme!

You can get this as package from hg.sr.ht/~arnebab/guile-define-typed.

```
(define-module (define-typed) #:export (define-typed* define-typed))

(import (srfi :11 let-values))

;; common procedures
(define-inlinable (takes-single-value? proc)
  (equal? '(1 0 #f) (procedure-minimum-arity proc)))

(define-inlinable (call-and-check-return-type proc ret?)
  (if ret? ;; #f means: do not check
      ;; get the result
      (let ((res (proc)))
        ;; typecheck the result
        (unless (ret? res)
          (error "type error: return value ~a does not match ~a"
                 res ret?))
        ;; return the result
        res)
      (proc)))

(define-inlinable
  (call-and-check-return-type/proc proc check-values)
  ;; get the result
  (let-values ((res (proc)))
    ;; typecheck the result
    (unless (check-values res)
      (error "type error: return values ~a do not match ~a"
             res check-values))
    ;; return the result
    (apply values res)))

(define-inlinable
  (call-and-check-return-type/multiple proc return-checkers)
  ;; get the result
  (let-values ((res (proc)))
    ;; typecheck the result
    (let loop ((check return-checkers) (r res))
      (when (pair? check)
        (unless ((car check) (car r))
          (error "type error: return values ~a do not match ~a"
                 res return-checkers))
        (loop (cdr check) (cdr r))))
    ;; return the result
    (apply values res)))
```

```
(define-inlinable (check-argument-and-type-count args types)
  (let loop ((a args) (t types))
    (unless (equal? (pair? a) (pair? t))
      (error "argument error: number of arguments ~a and types ~a differs"
             args types))
    (when (pair? a)
      (loop (cdr a) (cdr t)))))

(define (add-properties! proc name from-proc ret? types)
  ;; add procedure properties via an inner procedure
  (set-procedure-properties! proc (procedure-properties from-proc))
  ;; record the types
  (set-procedure-property! proc 'return-type ret?)
  (set-procedure-property! proc 'argument-types types)
  ;; preserve the name
  (set-procedure-property! proc 'name name))


;; specific to define-typed
(define-syntax check-types
  (syntax-rules ()
    ((_ (type? types? ...) (argument arguments ...))
     (begin
       (unless (type? argument)
         (error "type error ~a ~a" type? argument))
       (check-types (types? ...) (arguments ...))))
    ((_ () ()) #f)))


(define-syntax-rule (define-typed/base procname
                      (args ...) (types ...)
                      ret-proc ret-values
                      def lamb check ;; define or define*, ...
                      body ...)
  (begin
    (define properties-helper (lamb (args ...) body ...))
    (def (procname args ...)
         ;; create a sub-procedure to run after typecheck
         (define (inner)
           body ...)
         ;; typecheck the arguments
         (check (types ...) (args ...))
         ;; get and check the result
```

```
          (ret-proc inner ret-values))
     (check-argument-and-type-count
      (quote (args ...)) (quote (types ...)))
     ;; add properties and return the inner procedure
     (add-properties! procname 'procname properties-helper
                      ret-values (list types ...))))

;; helper without keyword support
(define-syntax-rule (define-typed/helper procname
                      (args ...) (types ...)
                      ret-proc ret-values
                      body ...)
  (define-typed/base procname
    (args ...) (types ...)
    ret-proc ret-values
    define lambda check-types ;; without keywords
    body ...))

;; Define a procedure with typechecks.
(define-syntax define-typed
  (syntax-rules ()
    ;; single checker: check all returned values via procedure
    ((_ (procname args ...) ((ret?) types ...)
        body ...)
     (define-typed/helper procname (args ...) (types ...)
       call-and-check-return-type/proc
       ret?
       body ...))
    ;; two or more return checkers: one per value (fixed number of
    ;; return values!)
    ((_ (procname args ...) ((ret1? ret2* ret*? ...) types ...)
        body ...)
     (begin
       (define return-checkers (list ret1? ret2* ret*? ...))
       (define-typed/helper procname (args ...) (types ...)
         call-and-check-return-type/multiple
         return-checkers
         body ...)))
    ;; single return checker: only check one value, further values are
    ;; discarded except if ret? is #f: then do not check, keep all
    ;; values
    ((_ (procname args ...) (ret? types ...)
        body ...)
     (define-typed/helper procname (args ...) (types ...)
```

```
       call-and-check-return-type
       ret?
       body ...))))


;; specific to define-typed*
(define-syntax check-types*
  (syntax-rules ()
    ((_ (type? types? ...) (argument arguments ...))
     (begin
       (if (and (keyword? type?)
                (keyword? argument))
           (unless (equal? type? argument)
             (error "Keywords in arguments and types differ ~a ~a"
                    type? argument))
           (unless (type? argument)
             (error "type error ~a ~a" type? argument)))
       (check-types* (types? ...) (arguments ...))))
    ((_ () ()) #f)))


;; helper with keyword support
(define-syntax-rule (define-typed*/helper procname
                      (args ...) (types ...)
                      ret-proc ret-values
                      body ...)
  (define-typed/base procname
    (args ...) (types ...)
    ret-proc ret-values
    define* lambda* check-types* ;; with keywords
    body ...))

;; Define a procedure with typecheck, taking keywords into acount like
;; define*.
(define-syntax define-typed*
  (syntax-rules ()
    ;; single checker: check all returned values via procedure
    ((_ (procname args ...) ((ret?) types ...)
        body ...)
     (define-typed*/helper procname (args ...) (types ...)
       call-and-check-return-type/proc
       ret?
       body ...))
    ;; two or more return checkers: one per value (fixed number of
```

```
    ;; return values!)
    ((_ (procname args ...) ((ret1? ret2* ret*? ...) types ...)
        body ...)
     (begin
       (define return-checkers (list ret1? ret2* ret*? ...))
       (define-typed*/helper procname (args ...) (types ...)
         call-and-check-return-type/multiple
         return-checkers
         body ...)))
    ;; single return checker: only check one value, further values are
    ;; discarded except if ret? is #f: then do not check, keep all
    ;; values
    ((_ (procname args ...) (ret? types ...)
        body ...)
     (define-typed*/helper procname (args ...) (types ...)
       call-and-check-return-type
       ret?
       body ...))))
```

This supports most features of regular define like docstrings, procedure properties, multiple values (thanks to Vivien!), and so forth.

`define-typed*` also supports keyword-arguments (thanks to Zelphir Kaltstahl's contracts), but is slower.

3 Benchmark

`define-typed` automates some of the guards of Optimizing Guile Scheme, so the compiler can optimize more (i.e. if you check for `real?`) but keep in mind that these checks are not free: use typechecks outside tight loops, except where they provably provide an improvement.

```
#!/usr/bin/env bash
exec guile -L . "$0"
; !#
(import (define-typed) (statprof))

(define (magnitude x y) (sqrt (+ (* x x) (* y y))))
(define (magnitude-handtyped x y)
  (unless (and (real? x) (inexact? x)
               (real? y) (inexact? y))
    (error "expected floats" x y))
  (sqrt (+ (* x x) (* y y))))
```

```
(define-inlinable (float? x)
  (and (real? x) (inexact? x)))
(define-inlinable (all-float? args)
  (not (member #f (map float? args))))

(define-typed (magnitude-typed x y) (#f float? float?)
  (sqrt (+ (* x x) (* y y))))

(define-typed
  (magnitude-typed/return x y)
  (float? float? float?)
  (sqrt (+ (* x x) (* y y))))

(define-typed
  (magnitude-typed/return-multiple x y)
  ((float? float?) float? float?)
  (values 1.0 (sqrt (+ (* x x) (* y y)))))

(define-typed
  (magnitude-typed/return-proc x y)
  ((all-float?) float? float?)
  (values 1.0 (sqrt (+ (* x x) (* y y)))))

(define-typed
  (magnitude-typed/return-lambda x y)
  (((λ (vals) (apply > vals))) number? number?)
  (values (sqrt (+ (* x x) (* y y))) x))

(define-typed*
  (magnitude-typed* x y #:key foo)
  (#f float? float? #:key not)
  (sqrt (+ (* x x) (* y y))))

(define-typed*
  (magnitude-typed*/return x y #:key foo)
  (float? float? float? #:key not)
  (sqrt (+ (* x x) (* y y))))

(define-typed*
  (magnitude-typed*/return-multiple x y #:key foo)
  ((float? float?) float? float? #:key not)
  (values 1.0 (sqrt (+ (* x x) (* y y)))))

(define-typed*
```

```
  (magnitude-typed*/return-proc x y #:key foo)
  ((all-float?) float? float? #:key not)
  (values 1.0 (sqrt (+ (* x x) (* y y)))))

(define (benchmark proc)
  (statprof
    (λ _
      (let lp ((i 0))
        (when (< i 20000000)
          (proc 3.0 4.0)
          (lp (+ i 1)))))))

(for-each benchmark
  (list
    magnitude
    magnitude-handtyped
    magnitude-typed
    magnitude-typed/return
    magnitude-typed/return-multiple
    magnitude-typed/return-proc
    magnitude-typed/return-lambda
    magnitude-typed*
    magnitude-typed*/return
    magnitude-typed*/return-multiple
    magnitude-typed*/return-proc))
```

Results:

```
%     cumulative   self
time   seconds     seconds  procedure
 88.89      2.64      2.35  benchmark.scm:6:0:magnitude
 11.11      0.29      0.29  %after-gc-thunk
  0.00      2.64      0.00  benchmark.scm:62:1
  0.00      0.29      0.00  anon #x26abd070
---
Sample count: 54
Total time: 2.644871952 seconds (2.308499722 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
100.00      0.78      0.78  benchmark.scm:7:0:magnitude-handtyped
  0.00      0.78      0.00  benchmark.scm:69:1
---
Sample count: 17
Total time: 0.776747387 seconds (0.613221055 seconds in GC)
%     cumulative   self
```

```
time   seconds     seconds  procedure
100.00      0.75      0.75  benchmark.scm:18:0:magnitude-typed
  0.00      0.75      0.00  benchmark.scm:76:1
---
Sample count: 16
Total time: 0.746165599 seconds (0.593782636 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 92.59      0.85      0.85  benchmark.scm:21:0:magnitude-typed/return
  7.41      0.92      0.07  benchmark.scm:83:1
---
Sample count: 27
Total time: 0.921354791 seconds (0.650629947 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 54.10      1.79      1.25  benchmark.scm:26:0:magnitude-typed/return-multiple
 22.95      2.32      0.53  benchmark.scm:90:1
 11.48      0.27      0.27  %after-gc-thunk
 11.48      0.27      0.27  benchmark.scm:13:0:#{% float?-procedure}#
  0.00      0.27      0.00  anon #x26abd070
---
Sample count: 61
Total time: 2.316873964 seconds (1.742421793 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 27.05      3.85      1.13  benchmark.scm:31:0:magnitude-typed/return-proc
 20.49      2.61      0.86  benchmark.scm:15:0:#{% all-float?-procedure}#
 20.49      1.72      0.86  ice-9/boot-9.scm:227:5:map1
 13.93      0.69      0.58  ice-9/boot-9.scm:222:2:map
  8.20      4.19      0.34  benchmark.scm:97:1
  4.10      0.17      0.17  benchmark.scm:13:0:#{% float?-procedure}#
  3.28      0.14      0.14  %after-gc-thunk
  2.46      0.10      0.10  list?
  0.00      0.14      0.00  anon #x26abd070
---
Sample count: 122
Total time: 4.188747795 seconds (2.995410552 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 60.19      3.71      2.68  benchmark.scm:56:0:magnitude-typed/return-lambda
 16.67      4.46      0.74  benchmark.scm:132:1
 14.81      0.66      0.66  >
  8.33      0.37      0.37  %after-gc-thunk
  0.00      0.37      0.00  anon #x26abd070
```

```
---
Sample count: 108
Total time: 4.457455449 seconds (3.702323067 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 64.18      2.75      1.79  benchmark.scm:36:0:magnitude-typed*
 34.33      0.96      0.96  %after-gc-thunk
  1.49      2.79      0.04  benchmark.scm:104:1
  0.00      0.96      0.00  anon #x26abd070
---
Sample count: 67
Total time: 2.792124506 seconds (2.408480193 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 64.79      3.05      1.98  benchmark.scm:41:0:magnitude-typed*/return
 33.80      1.03      1.03  %after-gc-thunk
  1.41      0.04      0.04  benchmark.scm:13:0:#{% float?-procedure}#
  0.00      3.05      0.00  benchmark.scm:111:1
  0.00      1.03      0.00  anon #x26abd070
---
Sample count: 71
Total time: 3.048783513 seconds (2.552226127 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 62.73      3.68      2.82  benchmark.scm:46:0:magnitude-typed*/return-multiple
 18.18      4.49      0.82  benchmark.scm:118:1
 14.55      0.65      0.65  %after-gc-thunk
  4.55      0.20      0.20  benchmark.scm:13:0:#{% float?-procedure}#
  0.00      0.65      0.00  anon #x26abd070
---
Sample count: 110
Total time: 4.493843352 seconds (3.663045745 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 38.41      5.68      2.37  benchmark.scm:51:0:magnitude-typed*/return-proc
 23.78      2.97      1.47  benchmark.scm:15:0:#{% all-float?-procedure}#
 11.59      1.77      0.72  ice-9/boot-9.scm:227:5:map1
  7.93      6.17      0.49  benchmark.scm:125:1
  5.49      0.45      0.34  ice-9/boot-9.scm:222:2:map
  5.49      0.34      0.34  %after-gc-thunk
  5.49      0.34      0.34  benchmark.scm:13:0:#{% float?-procedure}#
  1.83      0.11      0.11  list?
  0.00      0.34      0.00  anon #x26abd070
---
```

```
Sample count: 164
Total time: 6.173611964 seconds (4.77945362 seconds in GC)
```

`define-typed` reaches the performance of the hand-optimized procedure magnitude-handtyped while `define-typed*` is as fast as an untyped procedure in this case where constraining types provides a big benefit.

4 Summary

Typechecks from `define-typed` provide a type boundary that can help the compiler optimize instead of compile-time checked static typing.

You can do more advanced checks by providing your own test procedures and validating your API elegantly, but these may not help the compiler produce faster code.

> define-typed: a static type syntax-rules macro for Guile to create API contracts and help the JIT compiler create more optimized code.

But keep in mind that this does not actually provide static program analysis like while-you-write type checks. It's simply syntactic sugar for a boundary through which only allowed values can pass. Thanks to program flow analysis by the just-in-time compiler, it can make your code faster, but that's not guaranteed. It may be useful for your next API definition.