

Metaprogramming Haskell, The Racket Way

Alexis King

Northwestern University & PLT

```
#!/bin/bash
```

```
set -ueo pipefail
```

```
curl -s http://data-source.com/api/data.json \  
  | jq '.[] | { name: payload.name }' \  
  | python3 data-processor.py
```

```
#!/bin/bash
set -ueo pipefail

prefix_lines () {
    sed -e "s/^/$1: /"
}

with_prefix_outerr () {
    mk_pipe err_out err_in
    { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-
    prefix_lines stderr <&$err_in-
}

curl -s http://data-source.com/api/data.json \
    | jq '.[] | { name: payload.name }' \
    | with_prefix_outerr python3 data-processor.py \
    |& tee all-output.log \
    | grep -F '[info]'
```

```

import io, os, sys, threading
from subprocess import Popen, PIPE

def prefix_lines(prefix, unprefix_in, prefixed_out):
    def do_prefix_lines():
        for line in unprefix_in:
            prefixed_line = prefix + ': ' + line
            prefixed_out.write(prefixed_line)
    t = threading.Thread(target=do_prefix_lines, daemon=True)
    t.start()
    return t

def prefix_outerr(unprefix_out, unprefix_err,
                  prefixed_out, prefixed_err):
    t_out = prefix_lines('stdout', unprefix_out, prefixed_out)
    t_err = prefix_lines('stderr', unprefix_err, prefixed_err)
    return t_out, t_err

if __name__ == '__main__':
    curl = Popen(['curl', '-s', 'http://data-source.com/api/data.json'], stdout=PIPE)
    jq = Popen(['jq', '.[] | { name: payload.name }', stdin=curl.stdout, stdout=PIPE])

    with Pipe() as data_out, Pipe() as data_err:
        processor = threading.Thread(
            target=data_processor,
            args=(jq.stdout, data_out.output, data_err.output),
            daemon=True)
        processor.start()
        with Pipe() as prefixed_data:
            t_out, t_err = prefix_outerr(data_out.input, data_err.input,
                                         prefixed_data.output, prefixed_data.output)
            do_on_finish([t_out, t_err], lambda: prefixed_data.output.close())
            tee = Popen(['tee', 'all-output.log'], stdin=prefixed_data.input, stdout=PIPE)
            grep = Popen(['grep', '-F', '[info]'], stdin=tee.stdout, stdout=sys.stdout)
            sys.exit(grep.wait())

```

```
#!/bin/bash
set -ueo pipefail

prefix_lines () {
    sed -e "s/^/$1: /"
}

with_prefix_outerr () {
    mk_pipe err_out err_in
    { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-
    prefix_lines stderr <&$err_in-
}

curl -s http://data-source.com/api/data.json \
| jq '.[ ] | { name: payload.name }' \
| with_prefix_outerr python3 data-processor.py \
|& tee all-output.log \
| grep -F '[info]'
```

```
import io, os, sys, threading
from subprocess import Popen, PIPE

def prefix_lines(prefix, unprefix_in, prefixed_out):
    def do_prefix_lines():
        for line in unprefix_in:
            prefixed_line = prefix + ': ' + line
            prefixed_out.write(prefixed_line)
    t = threading.Thread(target=do_prefix_lines, daemon=True)
    t.start()
    return t

def prefix_outerr(unprefix_out, unprefix_err,
                  prefixed_out, prefixed_err):
    t_out = prefix_lines('stdout', unprefix_out, prefixed_out)
    t_err = prefix_lines('stderr', unprefix_err, prefixed_err)
    return t_out, t_err

if __name__ == '__main__':
    curl = Popen(['curl', '-s', 'http://data-source.com/api/data.json'], stdout=PIPE)
    jq = Popen(['jq', '.[ ] | { name: payload.name }', stdin=curl.stdout, stdout=PIPE])

    with Pipe() as data_out, Pipe() as data_err:
        processor = threading.Thread(
            target=data_processor,
            args=(jq.stdout, data_out.output, data_err.output),
            daemon=True)
        processor.start()
        with Pipe() as prefixed_data:
            t_out, t_err = prefix_outerr(data_out.input, data_err.input,
                                         prefixed_data.output, prefixed_data.output)

            do_on_finish([t_out, t_err], lambda: prefixed_data.output.close())
            tee = Popen(['tee', 'all-output.log'], stdin=prefixed_data.input, stdout=PIPE)
            grep = Popen(['grep', '-F', '[info]'], stdin=tee.stdout, stdout=sys.stdout)
            sys.exit(grep.wait())
```

```
with_prefix_outerr () {  
    mk_pipe err_out err_in  
    { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-  
    prefix_lines stderr <&$err_in-  
}
```

```
curl -s http://data-source.com/api/data.json \  
| jq '.[ ] | { name: payload.name }' \  
| with_prefix_outerr python3 data-processor.py \  
|& tee all-output.log \  
| grep -F '[info]'
```

```
with_prefix_outerr () {  
    mk_pipe err_out err_in  
    { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-  
    prefix_lines stderr <&$err_in-  
}  
  
curl -s http://data-source.com/api/data.json \  
| jq '.[ ] | { name: payload.name }' \  
| with_prefix_outerr python3 data-processor.py \  
|& tee all-output.log \  
| grep -F '[info]'
```

```
main.o: main.c defs.h  
    cc -c main.c  
kbd.o: kbd.c defs.h command.h  
    cc -c kbd.c  
command.o: command.c defs.h command.h  
    cc -c command.c  
display.o: display.c defs.h buffer.h  
    cc -c display.c  
clean:  
    rm edit main.o kbd.o command.o display.o
```

```
with_prefix_outerr () {
  mk_pipe err_out err_in
  { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-
  prefix_lines stderr <&$err_in-
}
```

```
curl -s http://data-source.com/api/data.json \
| jq '.[ ] | { name: payload.name }' \
| with_prefix_outerr python3 data-processor.py \
|& tee all-output.log \
| grep -F '[info]'
```

```
main.o: main.c defs.h
      cc -c main.c
kbd.o: kbd.c defs.h command.h
      cc -c kbd.c
command.o: command.c defs.h command.h
      cc -c command.c
display.o: display.c defs.h buffer.h
      cc -c display.c
clean:
      rm edit main.o kbd.o command.o display.o
```

```
\subsection{Haskell as Macros} \label{sub:hh-core}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see \texttt{hackett/private/kernel}).

The Hackett kernel language consists of just these pieces:

```
\begin{enumerate}
\item the core typechecker and core type language,
```



```
with_prefix_outerr () {
  mk_pipe err_out err_in
  { "$@" 2>&3- | prefix_lines stdout & } 3>&$err_out-
  prefix_lines stderr <&$err_in-
}
```

```
curl -s http://data-source.com/api/data.json \
| jq '.[ ] | { name: payload.name }' \
| with_prefix_outerr python3 data-processor.py \
|& tee all-output.log \
| grep -F '[info]'
```

```
main.o: main.c defs.h
      cc -c main.c
kbd.o: kbd.c defs.h command.h
      cc -c kbd.c
command.o: command.c defs.h command.h
      cc -c command.c
display.o: display.c defs.h buffer.h
      cc -c display.c
clean:
      rm edit main.o kbd.o command.o display.o
```

```
\subsection{Haskell as Macros} \label{sub:hh-core}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see \texttt{hackett/private/kernel}).

The Hackett kernel language consists of just these pieces:

```
\begin{enumerate}
\item the core typechecker and core type language,
```

```
server {
  listen      80;
  access_log  logs/domain1.access.log  main;
  root        html;

  location ~ /\.php$ {
    fastcgi_pass 127.0.0.1:1025;
  }
}
```

#lang rash

```
(define-syntax-rule (with-prefix-out+err block)
  (seq {
    |& #:with [pipe] seq {
      |& seq block |&> prefix-lines "stdout" err> pipe-out &bg
      |&> prefix-lines "stderr" in< pipe-in out> &err } })))

curl -s http://data-source.com/api/data.json \
| jq "[]" | { name: payload.name }" \
|& with-prefix-out+err { python3 data-processor.py } \
| tee all-output.log \
| grep -F "[info]"
```

```
main.o: main.c defs.h
      cc -c main.c
kbd.o: kbd.c defs.h command.h
      cc -c kbd.c
command.o: command.c defs.h command.h
      cc -c command.c
display.o: display.c defs.h buffer.h
      cc -c display.c
clean:
      rm edit main.o kbd.o command.o display.o
```

```
\subsection{Haskell as Macros} \label{sub:hh-core}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see \texttt{hackett/private/kernel}).

The Hackett kernel language consists of just these pieces:

```
\begin{enumerate}
```

```
\item the core typechecker and core type language,
```

```
server {
    listen      80;
    access_log  logs/domain1.access.log  main;
    root        html;

    location ~ /\.php$ {
        fastcgi_pass 127.0.0.1:1025;
    }
}
```

#lang rash

```
(define-syntax-rule (with-prefix-out+err block)
  (seq {
    |& #:with [pipe] seq {
      |& seq block |&> prefix-lines "stdout" err> pipe-out &bg
      |&> prefix-lines "stderr" in< pipe-in out> &err } })))

curl -s http://data-source.com/api/data.json \
| jq ".[] | { name: payload.name }" \
|& with-prefix-out+err { python3 data-processor.py } \
| tee all-output.log \
| grep -F "[info]"
```

(require make)

```
(make [("main.o" ["main.c" "defs.h"])]
      (cc "main.c"))
[("kbd.o" ["kbd.c" "defs.h" "command.h"])]
(cc "kbd.c"))
[("command.o" ["command.c" "defs.h" "command.h"])]
(cc "command.c"))
[("clean")]
(remove-files "main.o" "kbd.o" "command.o"))]
```

```
\subsection{Haskell as Macros} \label{sub:hh-core}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see \texttt{hackett/private/kernel}).

The Hackett kernel language consists of just these pieces:

```
\begin{enumerate}
\item the core typechecker and core type language,
```

```
server {
    listen      80;
    access_log  logs/domain1.access.log  main;
    root        html;

    location ~ /\.php$ {
        fastcgi_pass 127.0.0.1:1025;
    }
}
```

#lang rash

```
(define-syntax-rule (with-prefix-out+err block)
  (seq {
    |& #:with [pipe] seq {
      |& seq block |&> prefix-lines "stdout" err> pipe-out &bg
      |&> prefix-lines "stderr" in< pipe-in out> &err } })))

curl -s http://data-source.com/api/data.json \
| jq ".[] | { name: payload.name }" \
|& with-prefix-out+err { python3 data-processor.py } \
| tee all-output.log \
| grep -F "[info]"
```

(require make)

```
(make [("main.o" ["main.c" "defs.h"])]
      (cc "main.c"))
[("kbd.o" ["kbd.c" "defs.h" "command.h"])]
(cc "kbd.c")]
[("command.o" ["command.c" "defs.h" "command.h"])]
(cc "command.c")]
[("clean")]
(remove-files "main.o" "kbd.o" "command.o"])]
```

#lang scribble/acmart

```
@section[#:tag "hh-core"]{Haskell as Macros}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see `@tt{hackett/private/kernel}`).

The Hackett kernel language consists of just these pieces:

```
@itemlist[
  #:style 'ordered
  @item{the core typechecker and core type language,}]
```

```
server {
  listen      80;
  access_log  logs/domain1.access.log  main;
  root        html;

  location ~ /\.php$ {
    fastcgi_pass 127.0.0.1:1025;
  }
}
```

#lang rash

```
(define-syntax-rule (with-prefix-out+err block)
  (seq {
    |& #:with [pipe] seq {
      |& seq block |&> prefix-lines "stdout" err> pipe-out &bg
      |&> prefix-lines "stderr" in< pipe-in out> &err } })))

curl -s http://data-source.com/api/data.json \
| jq ".[] | { name: payload.name }" \
|& with-prefix-out+err { python3 data-processor.py } \
| tee all-output.log \
| grep -F "[info]"
```

(require make)

```
(make [("main.o" ["main.c" "defs.h"])]
      (cc "main.c")
      [("kbd.o" ["kbd.c" "defs.h" "command.h"])]
      (cc "kbd.c")
      [("command.o" ["command.c" "defs.h" "command.h"])]
      (cc "command.c")
      [("clean")
       (remove-files "main.o" "kbd.o" "command.o")]))
```

#lang scribble/acmart

```
@section[#:tag "hh-core"]{Haskell as Macros}
```

While Hackett implements most of the Haskell core language, it can shift a number of pieces from the core into programmer-defined libraries. Hackett's kernel language is not theoretical; it is defined as an actual Racket language, i.e., a module that exports syntactic forms and run-time functions (see `@tt{hackett/private/kernel}`).

The Hackett kernel language consists of just these pieces:

```
@itemlist[
  #:style 'ordered
  @item{the core typechecker and core type language,}]
```

(require web-server)

```
(define-values [dispatch get-url]
  (dispatch-rules
    [("") get-index]
    [("users" (id-param)) get-user-profile]))

(serve/servlet dispatch
  #:port 80
  #:log-file "logs/access.log"
  #:server-root-path "html")
```

```

#lang at-exp racket
(require rash make scribble/base web-server)

(define-values [dispatch get-url]
  (dispatch-rules
    [("docs") get-docs]
    [("build") #:method "post" post-build]))

(define (get-docs)
  (response/render
    @decode{
      @title{API Documentation}
      You can send a @tt{POST} request to
      @tt{/build} to trigger a build.}))

(define (post-build)
  (make (["processed-data.csv" ("raw-data.log")
        @rash{python3 process-data.py out> "processed-data.csv"}]
    ["raw-data.log" ("data-collector" "input-config.json")
        @rash{./data-collector input-config.json \
              out> "raw-data.log" err> "errors.log"}]
    ["data-collector" ("data-collector.c")
        @rash{gcc data-collector.c -o data-collector}])
    "processed-data.csv")
  (response/file "processed-data.csv"))

```



Racket

Macros

A Talk in Two Parts

I. A crash course in Racket macros.

- What makes Racket macros special?

II. A look at **Hackett**, and how it combines Racket macros with Haskell.

A brief introduction to Racket macros

Languages do not, in general, compose.

Languages do not, in general, compose.

Problem 1: Syntactic dissonance.

Racket's cop-out: give up, use s-expressions.

```
(define (prefix-lines prefix
                     unprefixed-in
                     prefixed-out)
  (define (do-prefix-lines)
    (for ([line (in-lines unprefixed-in)])
      (define prefixed-line
        (~a prefix ": " line))
      (displayln prefixed-line)))
  (thread do-prefix-lines))
```

```
(define-values [dispatch get-url]
  (dispatch-rules
    [( "home" )
     get-index]
    [( "old_stuff" _ ...)
     (make-redirect "new_stuff")]))
```

Languages do not, in general, compose.

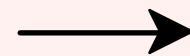
Problem 2: Semantic composition.

Solution: Define semantics via local rewriting.

```
(define-values [dispatch get-url]
  (dispatch-rules
    [("start-build" (string-arg))
     (lambda (request target-to-make)
       (make
        ([ "processed-data.csv" ("raw-data.log")
          (process-data "raw-data.log" "processed-data.csv")]
        ["raw-data.log" ("data-collector" "input-config.json")
          (collect-data "input-config.json" "raw-data.log")])
        target-to-make))]))
```

Macros are *local, code-to-code* transformations.

```
(match (f x)
  [(list fst snd)
   (println (+ fst snd))]
  [_
   #false])
```



```
(let ([tmp (f x)])
  (if (and (list? tmp)
           (= (length tmp) 2))
      (let ([fst (car tmp)]
            [snd (cadr tmp)])
        (println (+ fst snd)))
      #false))
```

```
(make (["out.txt" ("in.txt")
        (match (file->lines "in.txt")
          [(cons first-line other-lines)
           (display-lines-to-file
            (cons first-line (reverse other-lines))
            "out.txt")])])))
```



```
(make/proc
  (list (cons "out.txt" (list "in.txt")
    (lambda ()
      (match (file->lines "in.txt")
        [(cons first-line other-lines)
         (display-lines-to-file
          (cons first-line (reverse other-lines))
          "out.txt")])
      ))))
  (current-command-line-arguments))
```

```
(make/proc
  (list (cons "out.txt" (list "in.txt"))
    (lambda ()
      (match (file->lines "in.txt")
        [(cons first-line other-lines)
         (display-lines-to-file
          (cons first-line (reverse other-lines))
          "out.txt")]))))
  (current-command-line-arguments))
```



```
(make/proc
  (list (cons "out.txt" (list "in.txt"))
    (lambda ()
      (let ([tmp (file->lines "in.txt")])
        (if (and (list? tmp)
                  (>= (length tmp) 1))
            (let ([first-line (car tmp)]
                  [other-lines (cdr tmp)])
              (display-lines-to-file
               (cons first-line (reverse other-lines))
               "out.txt"))))
      ))
  (current-command-line-arguments))
```


Macros define composable *notations* via local rewrite rules.

They are recursively expanded *at compile-time*.



expand : Racket-Program -> Kernel-Racket-Program

(has macros)

(does not)

Kernel-Racket-Program ::=

variable

| (lambda (*id* ...) *expr* ... +)

| (if *expr expr expr*)

| (let ([*id expr*] ...) *expr* ... +)

| (set! *id expr*)

| ...

Macros

Simple idea, subtle in practice.

Lots of work done to handle the subtleties.

1. Lexical scope for macros (aka “hygiene”).

1986: E. Kholbecker, D. P. Friedman, M. Felleisen, and B. Duba.
Hygienic Macro Expansion.

1991: W. Clinger and J. Rees. Macros that Work.

1992: K. Dybvig. Syntactic abstraction in Scheme.

2002: M. Flatt. Composable and compilable macros.

2007: R. Culpepper, S. Tobin-Hochstadt, and M. Flatt.
Advanced Macrology and the Implementation of Typed Scheme.

2010: R. Culpepper and M. Felleisen. Debugging hygienic macros.

2012: M. Flatt, R. Culpepper, D. Darais, and R. B. Findler.
Macros that Work Together.

2013: M. Flatt. Submodules in racket.

2015: M. D. Adams. Towards the Essence of Hygiene.

2016: M. Flatt. Bindings as sets of scopes.

...and many others.

Macros

Simple idea, subtle in practice.

Lots of work done to handle the subtleties.

1. Lexical scope for macros (aka “hygiene”).
2. Predictable, reproducible compilation and phase separation.

1986: E. Kholbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion.

1991: W. Clinger and J. Rees. Macros that Work.

1992: K. Dybvig. Syntactic abstraction in Scheme.

2002: M. Flatt. Composable and compilable macros.

2007: R. Culpepper, S. Tobin-Hochstadt, and M. Flatt. Advanced Macrology and the Implementation of Typed Scheme.

2010: R. Culpepper and M. Felleisen. Debugging hygienic macros.

2012: M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that Work Together.

2013: M. Flatt. Submodules in racket.

2015: M. D. Adams. Towards the Essence of Hygiene.

2016: M. Flatt. Bindings as sets of scopes.

...and many others.

Macros

Simple idea, subtle in practice.

Lots of work done to handle the subtleties.

1. Lexical scope for macros (aka “hygiene”).
2. Predictable, reproducible compilation and phase separation.
3. Cooperating/communicating macros.

1986: E. Kholbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion.

1991: W. Clinger and J. Rees. Macros that Work.

1992: K. Dybvig. Syntactic abstraction in Scheme.

2002: M. Flatt. Composable and compilable macros.

2007: R. Culpepper, S. Tobin-Hochstadt, and M. Flatt. Advanced Macrology and the Implementation of Typed Scheme.

2010: R. Culpepper and M. Felleisen. Debugging hygienic macros.

2012: M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that Work Together.

2013: M. Flatt. Submodules in racket.

2015: M. D. Adams. Towards the Essence of Hygiene.

2016: M. Flatt. Bindings as sets of scopes.

...and many others.

Traditional macro systems are *just* about rewrite rules.

Racket goes further by allowing macros to *communicate*.

```
(define-adt Tree  
  (Leaf value)  
  (Node left right))
```

Traditional macro systems are *just* about rewrite rules.

Racket goes further by allowing macros to *communicate*.

```
(define-adt Tree  
  (Leaf value)  
  (Node left right))
```

```
(define (sum-tree t)  
  (match-adt Tree t  
    [(Leaf value)  
     value]  
    [(Node left right)  
     (+ (sum-tree left)  
        (sum-tree right))]))
```


Traditional macro systems are *just* about rewrite rules.

Racket goes further by allowing macros to *communicate*.

```
(define-adt Tree  
  (Leaf value)  
  (Node left right))
```

```
(define (sum-tree t)  
  (match-adt Tree t  
    [(Node left right)  
     (+ (sum-tree left)  
        (sum-tree right))]))
```

match-adt: missing case for 'Leaf'

Traditional macro systems are *just* about rewrite rules.

Racket goes further by allowing macros to *communicate*.

```
(define-adt Tree  
  (Leaf value)  
  (Node left right))
```

ctors: Leaf, Node

```
(define (sum-tree t)  
  (match-adt Tree t  
    [(Node left right)  
      (+ (sum-tree left)  
          (sum-tree right))]))
```

match-adt: missing case for 'Leaf'

Traditional macro systems are *just* about rewrite rules.

Racket goes further by allowing macros to *communicate*.

```
(define-adt Tree  
  (Leaf value)  
  (Node left right))
```

```
(define (sum-tree t)  
  (match-adt Tree t  
    [(Node left right)  
     (+ (sum-tree left)  
        (sum-tree right))]))
```

match-adt: missing case for 'Leaf'

This is enormously powerful!

More information is more expressive power.

Lexical region sensitivity (e.g. 'this').

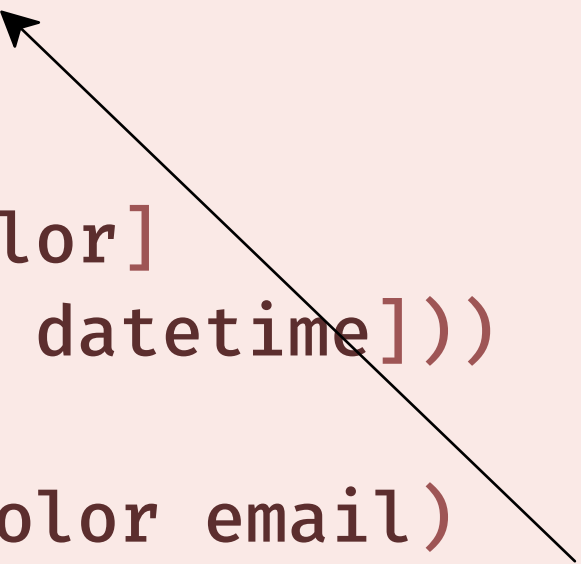
```
(class object%  
  (super-new)  
  (define/private (internal-beep)  
    (println "beep!"))  
  (define/public (beep)  
    (send this internal-beep)))
```

More information is more expressive power.

Generic programming (e.g. SQL generation).

```
(define-sql-enum color
  [red orange yellow green blue purple])
(define-sql-struct user
  ([email : string]
   [name : string]
   [favorite-color : color]
   [registration-date : datetime]))

(define (get-favorite-color email)
  (SELECT u.favorite-color FROM [u : user]
    WHERE (= u.email email)))
```

A black arrow originates from the 'user' struct definition in the first block and points to the '[u : user]' table reference in the SQL query of the second block.

More information is more expressive power.

Macro-extensible macros (e.g. pattern macros).

```
(define-pattern-macro
  (form-data [key-pat val-pat] ...)
  (list-no-order
    (binding:form key val-pat)
    ...))

(define (handle-form-submit data)
  (match data
    [(list-no-order
      (binding:form "action" "log-in")
      (binding:form "email" (? valid-email? email))
      (binding:form "password" password))
     (session-login! email password)]))
```

Not quite so local anymore!

Racket gets a lot of mileage out of its macro system.

Let's recap.

1. Macros define domain-specific *notations*.
2. They do this via *local rewrite rules*.
3. These rules are defined and applied *at compile-time*.
4. Racket supports querying the compile-time environment to enable *macro communication*.

Racket's macros are good!

But so is Haskell's type system.

I want both.

Hackett

(Haskell + Racket)

Demo

Algebraic Datatypes

```
data Maybe a = Nothing | Just a  
  deriving (Eq, Show)
```

```
(data (Maybe a) Nothing (Just a)  
  #:deriving [Eq Show])
```

Pattern Matching

```
case stringSplit "," str of
  [a, b] -> Point <$> fromParam a
              <*> fromParam b
  _ -> Left ("bad point: " ++ show str)
```

```
(case (string-split "," str)
  [(List a b) {Point <$> (from-param a)
                      <*> (from-param b)}]
  [_ (Left {"bad point: " ++ (show str)})])
```

Do Notation

```
do x <- [1, 2]
   y <- [3, 4]
   z <- [5, 6]
   pure (x, y, z)
```

```
(do [x <- (List 1 2)]
    [y <- (List 3 4)]
    [z <- (List 5 6)]
    (pure (Tuple x y z)))
```

Typeclasses

```
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> b      = b
  a      <> Nothing = a
  Just a  <> Just b  = Just (a <> b)
instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

```
(instance (forall [a] (Semigroup a) => (Semigroup (Maybe a)))
  [++ (λ* [[Nothing b] b]
          [[a Nothing] a]
          [[(Just a) (Just b)] (Just {a ++ b})]])])
(instance (forall [a] (Semigroup a) => (Monoid (Maybe a)))
  [mempty Nothing])
```

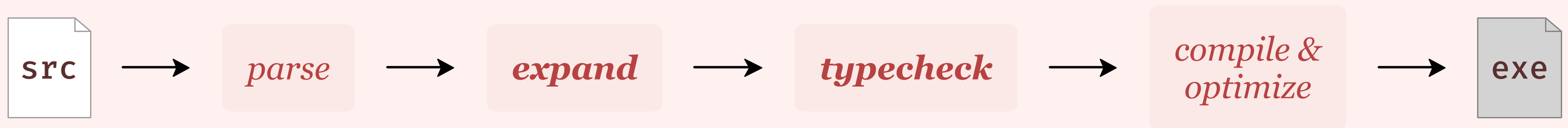
Haskell is really big.

Let's just focus on the macros.

How do we combine types and macros?



Idea: just expand first, then typecheck.



Will this work?

Yes! ...mostly.

This is the approach taken by Typed Racket.

A simple idea: typechecking macros is hard.
Therefore, expand first, then typecheck.

```
(match e
  [(list x y)
   (+ (* x 2) y)])
```



```
(let ([tmp e])
  (if (and (list? tmp) (= (length tmp) 2))
      (let ([x (car tmp)]
            [y (cadr tmp)])
        (+ (* x 2) y))
      (match-error)))
```

Only need to handle kernel language,
which is small, and can handle all macros!

Advanced Macrology and the Implementation of Typed Scheme

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Sam Tobin-Hochstadt
Northeastern University
samth@ccs.neu.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

PLT Scheme provides an expressive programming language implementation framework in order to enable experimentation with language design. This framework is rooted in PLT Scheme's hygienic macro system, but it has grown to encompass features that extend its capabilities beyond that of traditional macro systems.

In this paper we describe the features of PLT Scheme's language framework and demonstrate their use with a case study. Specifically, we present the design and implementation of Typed Scheme using the advanced language construction features of PLT Scheme.

1. Defining Languages

Since their creation, Lisp and Scheme macros have been used by programmers to extend their programming languages with notational abbreviations and domain-specific syntactic forms. Macros thus make it easier to read and write programs by bringing the programming language closer to the problem domain.

Discussions of macros often leave out the challenges that arise when macros need to work with other macros [6]. These challenges also appear in the construction of tools such as debuggers and static analyzers for a language defined via macros. The language tools should operate at the language's level of abstraction, not at the level of the underlying Scheme code. In general, collaborating macros and proper language abstraction require features from the macro system beyond those needed for isolated abstractions.

Collaborating macros must share information. For example, many syntactic forms in PLT Scheme deal with named structure types. The macro that defines new structure types publishes relevant information for other forms to consume. The pattern matching form [23] uses that information to check the arity of structure patterns and compile the pattern matching code.

Ideally, a similar sort of communication should take place between macros and language tools. When a new language can implement its language constructs in terms of corresponding constructs in the host language—for example, lexical scoping in the new language can be translated to lexical scoping in the host language—then the tools that analyze those constructs are reusable as well. For example, DrScheme's Check Syntax and debugging tools [3] work with the Algol60 [14], Lazy Scheme [1], Typed Scheme [20, 21],

and other translation-based languages [8] because of this kind of linguistic reuse.

Even in the ideal case, though, language tools require hints from the macro expansion process to interpret expanded programs and correlate the expanded code with the original source. Virtually every part of DrScheme [10], from its debugging support to its graphical analysis displays, benefits from the automatic source location information stored in syntax objects, adapted from the recommendations of the **syntax-case** report [7]. Source tracking is mostly automatic. The Stepper [2], on the other hand, must reconstruct or “unexpand” terms in the supported languages. For example, it must determine whether a chain of **if** expressions came from a **cond** expression, a **case** expression, an **or** expression, or something else. This reconstruction requires communication between the supported languages and the Stepper.

This paper presents the features of the language infrastructure that support collaborating macros, language definitions, and analysis tools. It illustrates the design and the pragmatics of these features with the implementation of Typed Scheme, a new member of the PLT Scheme language suite.

The paper is organized as follows. Section 2 introduces Typed Scheme, an extension of the PLT Scheme language with types. Section 3 presents the syntax framework that provides the foundation for macro programming and language implementation in PLT Scheme. Section 4 sketches the Typed Scheme implementation approach at the level of single definitions and expressions. Section 5 shows how the implementation scales up to multi-module programs. Finally, Section 6 discusses related work.

2. Typed Scheme

Typed Scheme is an explicitly-typed dialect of PLT Scheme designed to support the addition of types to existing programs one module at a time [21]. Typed modules interact with untyped modules safely and intuitively. The type system accommodates the idioms of Scheme programming, minimizing the need to change the structure of the program when adding types.¹ It supports the features of PLT Scheme, most importantly macros, modules, and structures.

2.1 The language

The syntax of Typed Scheme differs from its binding and definition

Of course, it's too good to be true.

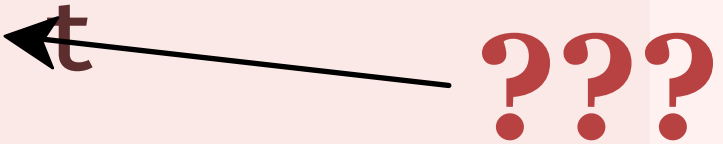
Hard to typecheck expansion of complicated macros.
(Most type inference schemes are incomplete; depend on manual intervention.)

Makes direct type-macro interaction impossible.
(Macros are gone by the time types exist.)

Type-Directed Macros

Remember match-adt?

```
(match-adt Tree t  
  [(Leaf value)  
   value]  
  [(Node left right)  
   (+ (sum-tree left)  
       (sum-tree right))])
```

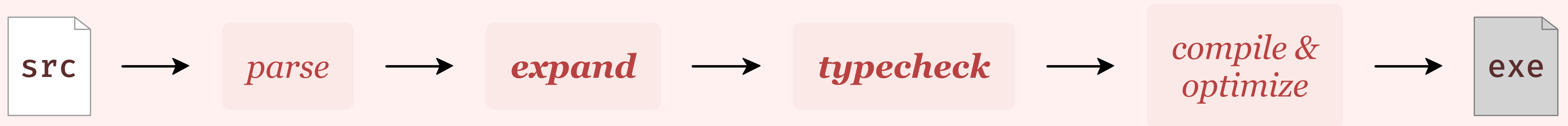


```
case t of  
  Leaf value ->  
    value  
  Node left right ->  
    sumTree left +  
    sumTree right
```

Haskell gets to use its types for good.

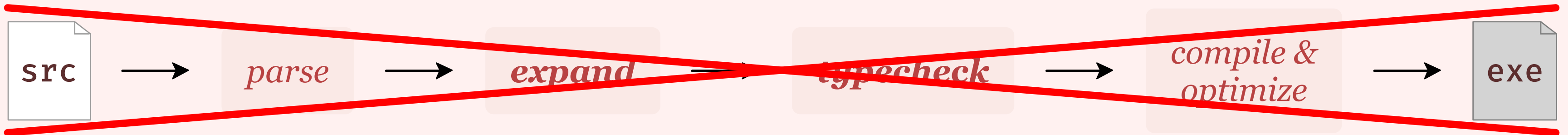
No type information means macros
become second-class citizens.

How can we fix this?



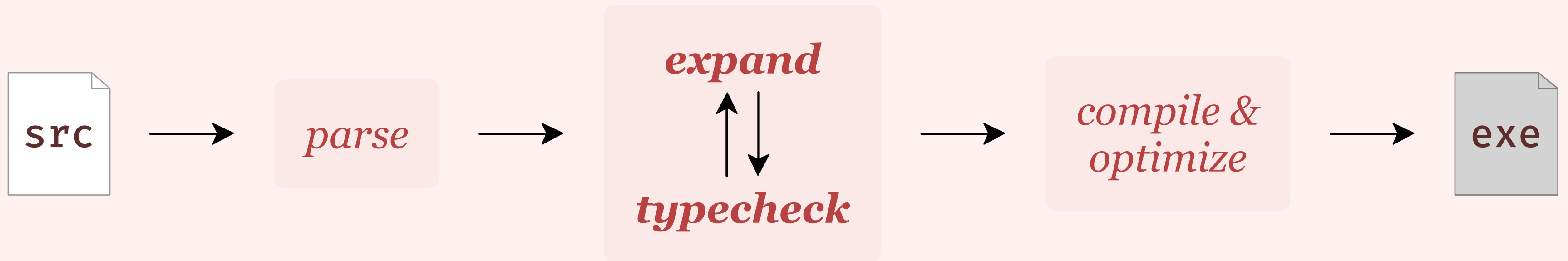
No type information means macros
become second-class citizens.

How can we fix this?



No type information means macros
become second-class citizens.

How can we fix this?



Answer: *interleave* typechecking and macroexpansion.

New problem: that's really hard.

Solution: let someone figure it out for you.

Chang, Knauth, and Greenman save the day!
Their trick: encode typechecking in the macro system.

```
; [LAM]
(define-typed-syntax (λ ([x:id : τin:type] ...) e) >>
  [[x >> x- : τin.norm] ... ⊢ e >> e- ⇒ τout])
-----
[⊢ (λ- (x- ...) e-) ⇒ (⇒ τin.norm ... τout)]])

; [APP]
(define-typed-syntax (%app efn earg ...) >>
  [⊢ efn >> efn- ⇒ (∼⇒ τin ... τout)]
  [⊢ earg >> earg- ⇐ τin] ...
-----
[⊢ (%app- efn- earg- ...) ⇒ τout]])
```

Key idea: every macro does both typechecking *and* desugaring, which together form *type erasure*.

Type Systems as Macros

Stephen Chang Alex Knauth Ben Greenman
PLT @ Northeastern University, Boston, MA, USA
{stchang,alexknauth,types}@ccs.neu.edu



Abstract

We present TURNSTILE, a metalanguage for creating typed embedded languages. To implement the type system, programmers write type checking rules resembling traditional judgment syntax. To implement the semantics, they incorporate elaborations into these rules. TURNSTILE critically depends on the idea of linguistic reuse. It exploits a *macro system* in a novel way to *simultaneously* type check and rewrite a surface program into a target language. Reusing a macro system also yields modular implementations whose rules may be mixed and matched to create other languages. Combined with typical compiler and runtime reuse, TURNSTILE produces performant typed embedded languages with little effort.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Specialized application languages

Keywords macros, type systems, typed embedded DSLs

1. Typed Embedded Languages

As Paul Hudak asserted, “we really don’t want to build a programming language from scratch ... better, let’s inherit the infrastructure of some other language” [23]. Unsurprisingly, many modern languages support the creation of such embedded languages [3, 18, 20, 22, 24–26, 41, 43, 46].

Programmers who wish to create *typed* embedded languages, however, have more limited options. Such languages typically reuse their host’s type system but, as a prominent project [45] recently remarked, this “*confines* them to (that) type system.” Also, reusing a type system may not create proper abstractions, e.g., type errors may be reported in host language terms. At the other extreme, a programmer can implement a type system from scratch [42], expending considerable effort and passing up many of the reuse benefits that embedding a language promises in the first place.

We present an alternative approach to implementing typed embedded languages. Rather than reuse a type system, we embed a type system in a host’s *macro system*. In other words, type checking is computed as part of macro expansion. Such an embedding fits naturally since a typical type checking algorithm traverses a surface program, synthesizes information from it, and then uses this information to rewrite the program, if it satisfies certain conditions, into a target language. *This kind of algorithm exactly matches the ideal use case for macros*. From this perspective, a type checker resembles

bles a special instance of a macro system and our approach exploits synergies resulting from this insight.

With our macro-based approach, programmers may implement a wide range of type rules, yet they need not create a type system from scratch since they may reuse components of the macro system itself for type checking. Indeed, programmers need only supply their desired type rules in an intuitive mathematical form. Creating type systems with macros also fosters robust linguistic abstractions, e.g., they report type errors with surface language terms. Finally, our approach produces naturally modular type systems that dually serve as libraries of mixable and matchable type rules, enabling further linguistic reuse [27]. When combined with the typical reuse of the runtime that embedded languages enjoy, our approach inherits the performance of its host and thus produces practical typed languages with significantly reduced effort.

We use Racket [12, 15], a production-quality Lisp and Scheme descendant, as our host language since Lisps are already a popular platform for creating embedded languages [17, 20]. Racket’s macro system in particular continues to improve on its predecessors [14] and has even influenced macro system design in modern non-Lisp languages [6, 8, 10, 48]. Thus programmers have created Racket-embedded languages for accomplishing a variety of tasks such as book publishing [7], program synthesis [44], and writing secure shell scripts [32].

The first part of the paper (§2-3) demonstrates a connection between type rules and macros by reusing Racket’s macro infrastructure for type checking in the creation of a typed embedded language. The second part (§4) introduces TURNSTILE, a metalanguage that abstracts the insights and techniques from the first part into convenient linguistic constructs. The third part (§5-7) shows that our approach both accommodates a variety of type systems and scales to realistic combinations of type system features. We demonstrate the former by implementing fifteen core languages ranging from simply-typed to F_{ω} , and the latter with the creation of a full-style type classes.

2. Creating Embedded Languages in Racket

This section summarizes the creation of embedded languages with Racket. Racket is not a single language but rather an ecosystem with which to create languages [12]. Racket code is organized into modules, e.g. LAM:¹

```
#lang racket
(define-m (lm x e) (λ (x) e))
(provide lm)
```

138

LAM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL’17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...\$15.00
<http://dx.doi.org/10.1145/3009837.3009886>

¹ Code note: For clarity and consistency, its examples use the following convention:

Type Systems as Macros is extremely clever.

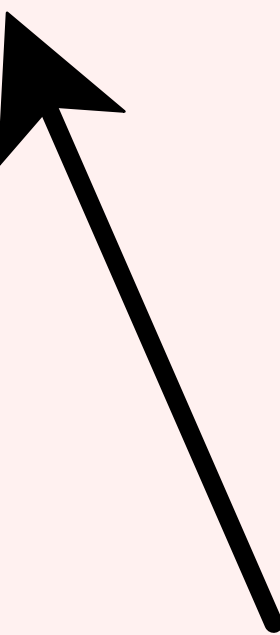
Can we scale it to a full language?

Challenges

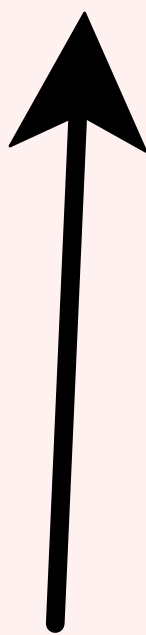
1. Designed to support languages with *local* inference.
2. Mostly tested on small languages (e.g. STLC, System F, etc.).
3. Too powerful: arbitrary (potentially unsound) type rules.

A Fundamental Tension

Local *vs.* Global



macros



types

In Haskell, type information can flow *backwards*.

```
(let ([x mempty])  
  {x ++ "foo"})
```




```
mempty : (forall [a] (Monoid a) => a)  
x : String  
t13^ = String
```

How does this cause trouble for macros?

Type-Directed Macros

Some macros want to look at type information.

```
(case t
  [(Leaf value)
   value]
  [(Node left right)
   (+ (sum-tree left)
      (sum-tree right))])
```



DM between case and typechecker.

case

What is the type of **t**?

(I hope it's something like (Tree Integer)!)

typechecker

The type I know for **t** is **t29**[^].

case

>:(

We've ended up in an awkward knot.

Global type inference means type information
sometimes propagates “backwards.”

We have to expand macros to learn their types.

`case` depends on the type of `t` to expand...
...but we need to expand `case` to learn the type of `t`.

But Haskell already deals with this problem!

Due to overloading, Haskell's semantics must be *Church-style* .

(That is, types affect the meaning of the program.)

`{mempty :: (Maybe Unit)}` $\xrightarrow{\text{eval}}$ `Nothing`

But Haskell has type-erasure, so how does this really work?


Answer: *elaboration.*

`{mempty :: String}` \longrightarrow `memptyString`

A type-directed, global rewriting step.

```
(def when
  (λ (condition value)
    (if condition
        value
        mempty)))

(do [date <- current-date]
  (pure
    {"[" ++ (date->string date) ++ "]" "
    ++ (basic-info->string info)
    ++ (when verbose?
          {" " ++ (extra-info->string info)}))}))
```



Secret sauce: *constraints*.[†]

```
mempty : (∀ [a] (Monoid a) => a)
```

```
(def when : (∀ [a] (Monoid a) =>
              {Boolean -> a -> a})
  (λ (memptya condition value)
    (if condition
        value
        memptya)))
```

```
(when memptyt32^ verbose? "extra info")
```

[†] Wadler and Blott, 1988

(when `memptyt32` verbose? "extra info")



(when `memptystring` verbose? "extra info")

This is *elaboration*.

Elaboration

- 1.** Constraint generation.
- 2.** Constraint solving.
- 3.** Type-directed program transformation.

Idea: leverage elaboration for type-directed macros.

Expand macros in multiple passes;
give them access to the constraint solver.

```
(#%delay-expression
```

```
t29^
```

```
(case t
```

```
[(Leaf value)  
value]
```


```
[(Node left right)
```

```
(+ (sum-tree left)
```

```
(sum-tree right))]))
```



```
t29^
```

1. Extensible constraint language.
2. Extensible constraint solver.
3. Elaborator macros. 

Lots of implementation challenges:
performance, ease of use, good error reporting.

Summary

Racket provides support for DSLs to allow mixing/matching composable notations.

One of the biggest features supporting DSLs is the Racket macro system.

Macros' expressive power is multiplied by access to compile-time information.

Synthesizing types and macros creates a system bigger than the sum of its parts.

We can leverage the Haskell constraint solver to do it, via multi-pass macroexpansion.