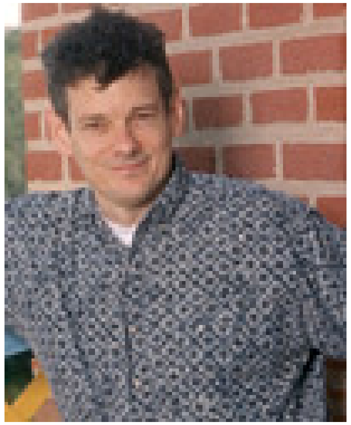
From Soft Scheme to Typed Scheme: 20 Years of *Scripts-to-Program* Conversion

Matthias Felleisen, PLT & NU PRL



Robert “Corky” Cartwright

User-Defined Data Types as an
Aid to Verifying LISP Programs
ICALP 1976, 228-256
Editors: Michaelson and Milner

verification

1975-77: Cartwright: convert lisp to typed lisp verification | 1975-77: Cartwright: convert lisp to typed lisp

wide-spectrum | 1981-91: Fagan: infer types for all *functional* Scheme programs

verification

wide-spectrum

rapid & extensible prototyping

1975-77: Cartwright: convert lisp to typed lisp

1981-91: Fagan: infer types for all *functional* Scheme programs

1991-94: Wright: HM-infer types for *all* Scheme programs

verification

wide-spectrum

rapid & extensible prototyping

1975-77: Cartwright: convert lisp to typed lisp

1981-91: Fagan: infer types for all *functional* Scheme programs

1991-94: Wright: HM-infer types for *all* Scheme programs

1994-98: Flanagan: *SBA-infer* types for Scheme components

verification	1975-77: Cartwright: convert lisp to typed lisp
wide-spectrum	1981-91: Fagan: infer types for all *functional* Scheme programs
	1991-94: Wright: HM-infer types for *all* Scheme programs
rapid & extensible prototyping	1994-98: Flanagan: *SBA-infer* types for Scheme components
	1996-98: Krishnamurthi: HM vs SBA/CFA

verification

wide-spectrum

rapid & extensible prototyping

1975-77: Cartwright: convert lisp to typed lisp

1981-91: Fagan: infer types for all *functional* Scheme programs

1991-94: Wright: HM-infer types for *all* Scheme programs

1994-98: Flanagan: *SBA-infer* types for Scheme components

1996-98: Krishnamurthi: HM vs SBA/CFA

1999-2006: Meunier: SBA-infer; *explicit* module *interfaces*

verification

wide-spectrum

rapid & extensible prototyping

1975-77: Cartwright: convert lisp to typed lisp

1981-91: Fagan: infer types for all *functional* Scheme programs

1991-94: Wright: HM-infer types for *all* Scheme programs

1994-98: Flanagan: *SBA-infer* types for Scheme components

1996-98: Krishnamurthi: HM vs SBA/CFA

1999-2006: Meunier: SBA-infer; *explicit* module *interfaces*

2005-2009: Tobin-Hochstadt: *Scripts to Programs*

verification	1975-77: Cartwright: convert lisp to typed lisp
wide-spectrum	1981-91: Fagan: infer types for all *functional* Scheme programs
rapid & extensible prototyping	1991-94: Wright: HM-infer types for *all* Scheme programs
	1994-98: Flanagan: *SBA-infer* types for Scheme components
	1996-98: Krishnamurthi: HM vs SBA/CFA
	1999-2000: Meunier: occurrence types for ISL
	1999-2002: Findler: contracts for Scheme
	1999-2006: Meunier: SBA-infer; *explicit* module *interfaces*
from scripts to programs	2005-2009: Tobin-Hochstadt: *Scripts to Programs*

verification

1975-77: Cartwright: convert lisp to typed lisp

wide-spectrum

1981-91: Fagan: infer types for all *functional* Scheme programs

1991-94: Wright: HM-infer types for *all* Scheme programs

rapid & extensible prototyping

1994-98: Flanagan: *SBA-infer* types for Scheme components

1996-98: Krishnamurthi: HM vs SBA/CFA

1999-2000: Meunier: occurrence types for ISL

1999-2002: Findler: contracts for Scheme

1999-2006: Meunier: SBA-infer; *explicit* module *interfaces*

from scripts to programs

2005-2009: Tobin-Hochstadt: *Scripts to Programs*

2008-2011: Stevie Strickland: Typed **PLT** Scheme (class.ss)

Cartwright 1976

The Dream: Write programs now; verify them later.

Cartwright 1976

The Dream: Write programs now; verify them later.

functional LISP >>> imperative Algol

Cartwright 1976

The Dream: Write programs now; verify them later.

functional LISP >>> imperative Algol

first add types, then prove theorems

Mike Fagan 1981-1987

Realize the Dream, at least the Types Part

Mike Fagan 1987-1991

Soft Typing: Infer Types for *All* Functional Scheme Programs

Mike Fagan 1981-1987

Realize the Dream, at least the Types Part

Mike Fagan 1987-1991

Soft Typing: Infer Types for *All* Functional Scheme Programs

turn true recursive unions into ML's datatype;
see Henglein's work in 1990s

Mike Fagan 1981-1987
Realize the Dream, at least the Types Part

Mike Fagan 1987-1991

Soft Typing: Infer Types for *All* Functional Scheme Programs

turn true recursive unions into ML's datatype;
see Henglein's work in 1990s

turn true recursive unions into Remy's record algebra;
use HM type inference to restore types
with *slack variables* for catching mismatches

```
(define value
  (letrec ((get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ((x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                            (car (regexp-match "[A-Z/]+" name))))))
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
           l))))
```

$s = (t \rightarrow \underline{int})$

$t = (v \rightarrow \underline{char})$

$v = \underline{double}$

unification
~
Gaussian elim

```scheme
(define value
  (letrec ((get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ((x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                           (car (regexp-match "[A-Z/]+" name))))))
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
        l))))
```

$s = (t \rightarrow \underline{int})$

$t = (v \rightarrow \underline{char})$

$v = \underline{double}$

unification ~ Gaussian elim

$s \subseteq \{ \text{dom} : t, \text{rng} : \text{int} \} \text{ u } \{ \text{num} : 0 \}$

$t \subseteq \{ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0 \}$

$v \subseteq \underline{double}$

$s = \{ \text{dom} : t, \text{rng} : \text{int} \} \text{ u } \{ \text{num} : 0 \} \text{ u } \gamma$

$t = \{ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0 \} \text{ u } \delta$

$v = \underline{double} \text{ u } \epsilon$

unification ~ Gaussian elim slack filling

```
(define value
  (letrec ((get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ((x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                           (car (regexp-match "[A-Z/]+" name))))))
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
       l))))
```

$s = (t \rightarrow \underline{\text{int}})$
$t = (v \rightarrow \underline{\text{char}})$
$v = \underline{\text{double}}$

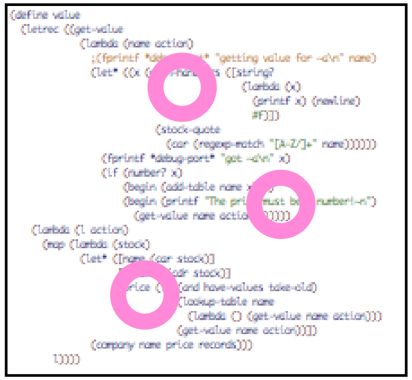
unification
~
Gaussian elim

$s \subseteq \{\ \text{dom} : t, \text{rng} : \text{int}\ \} \cup \{\ \text{num} : 0\ \}$
$t \subseteq \{\ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0\ \}$
$v \subseteq \underline{\text{double}}$

$s = \{\ \text{dom} : t, \text{rng} : \text{int}\ \} \cup \{\ \text{num} : 0\ \} \cup \gamma$
$t = \{\ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0\ \} \cup \delta$
$v = \underline{\text{double}} \cup \epsilon$

unification
~
Gaussian elim
slack filling

recursive domain of values ~ recursive union type



$$s = (t \rightarrow \underline{\text{int}})$$
$$t = (v \rightarrow \underline{\text{char}})$$
$$v = \underline{\text{double}}$$

unification
~
Gaussian elim

$$s \subseteq \{\ \text{dom} : t, \text{rng} : \text{int}\ \} \text{ u } \{\ \text{num} : 0\ \}$$
$$t \subseteq \{\ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0\ \}$$
$$v \subseteq \underline{\text{double}}$$

$$s = \{\ \text{dom} : t, \text{rng} : \text{int}\ \} \text{ u } \{\ \text{num} : 0\ \} \text{ u } \gamma$$
$$t = \{\ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0\ \} \text{ u } \delta$$
$$v = \underline{\text{double}} \text{ u } \epsilon$$

unification
~
Gaussian elim
slack filling

recursive domain of values ~ recursive union type

```
;; RussianDoll = 'doll u (cons RussianDoll empty)

;; RussianDoll -> Nat
(define (depth rd)
  (cond
    [(symbol? rd) 0]
    [else (+ 1 (depth (car rd)))]))
```

;; Fagan's "soft typer" can confirm the comments:

```
;; RussianDoll = 'doll u (cons RussianDoll empty)

;; RussianDoll -> Nat
(define (depth rd)
  (cond
    [(symbol? rd) 0]
    [else (+ 1 (depth (car rd)))]))
```

```
;; Prep = True u False u (Boolean -> Prep)

;; Prep -> Boolean
(define (taut p)
  (cond
    [(boolean? p) p]
    [else (and (taut (p true)) (taut (p false)))]))
```

;; Fagan's "soft typer" can also confirm these comments:

```
;; Prep = True u False u (Boolean -> Prep)

;; Prep -> Boolean
(define (taut p)
  (cond
    [(boolean? p) p]
    [else (and (taut (p true)) (taut (p false)))]))
```

Andrew Wright 1991-1994

Soft Scheme

problem: Fagan can deal with nothing but toy programs

Andrew Wright 1991-1994

Soft Scheme

problem: Fagan can deal with nothing but toy programs

solution: improve implementation algebra;
cope with mutations, continuations, etc.

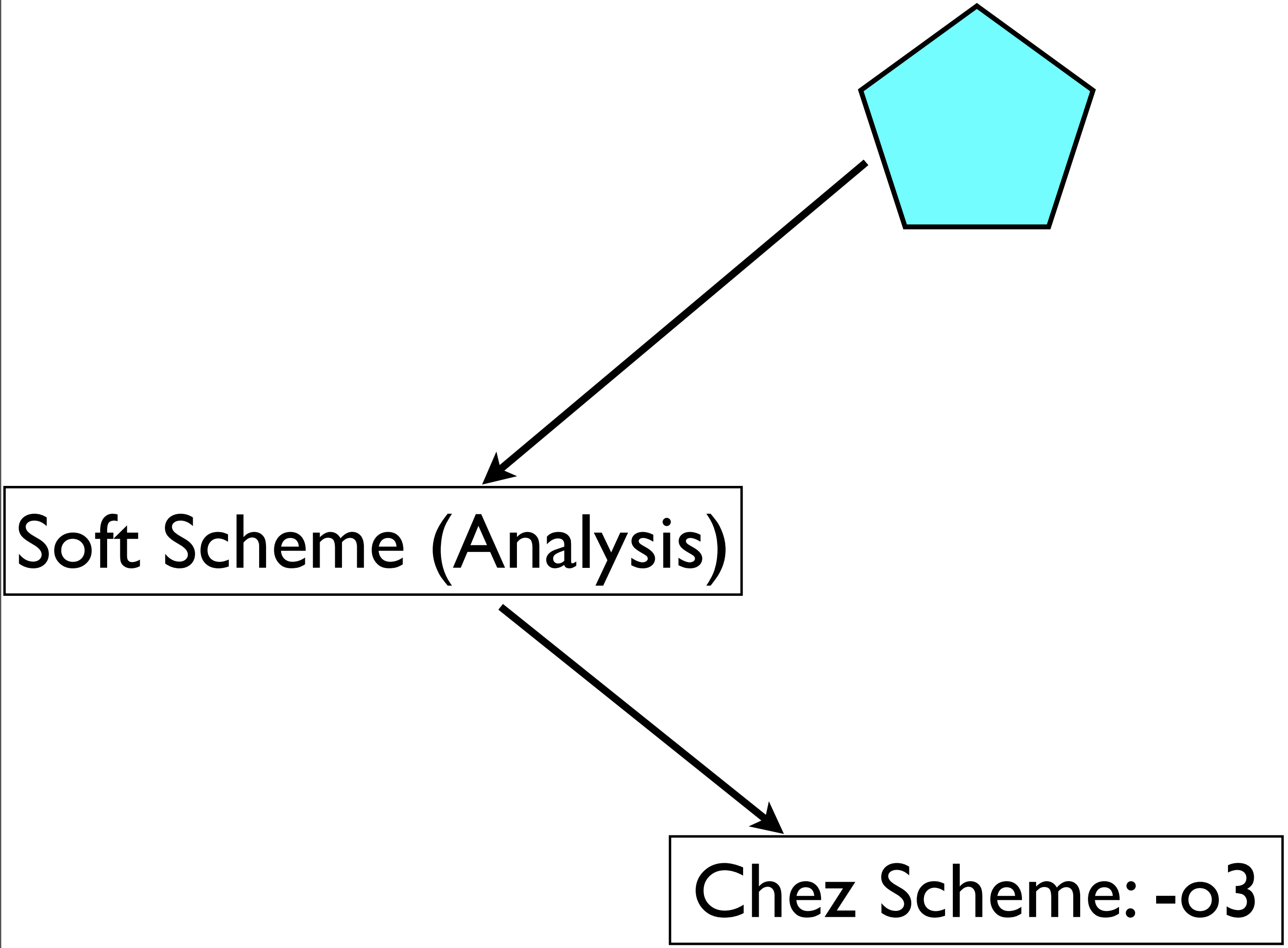
Andrew Wright 1991-1994

Soft Scheme

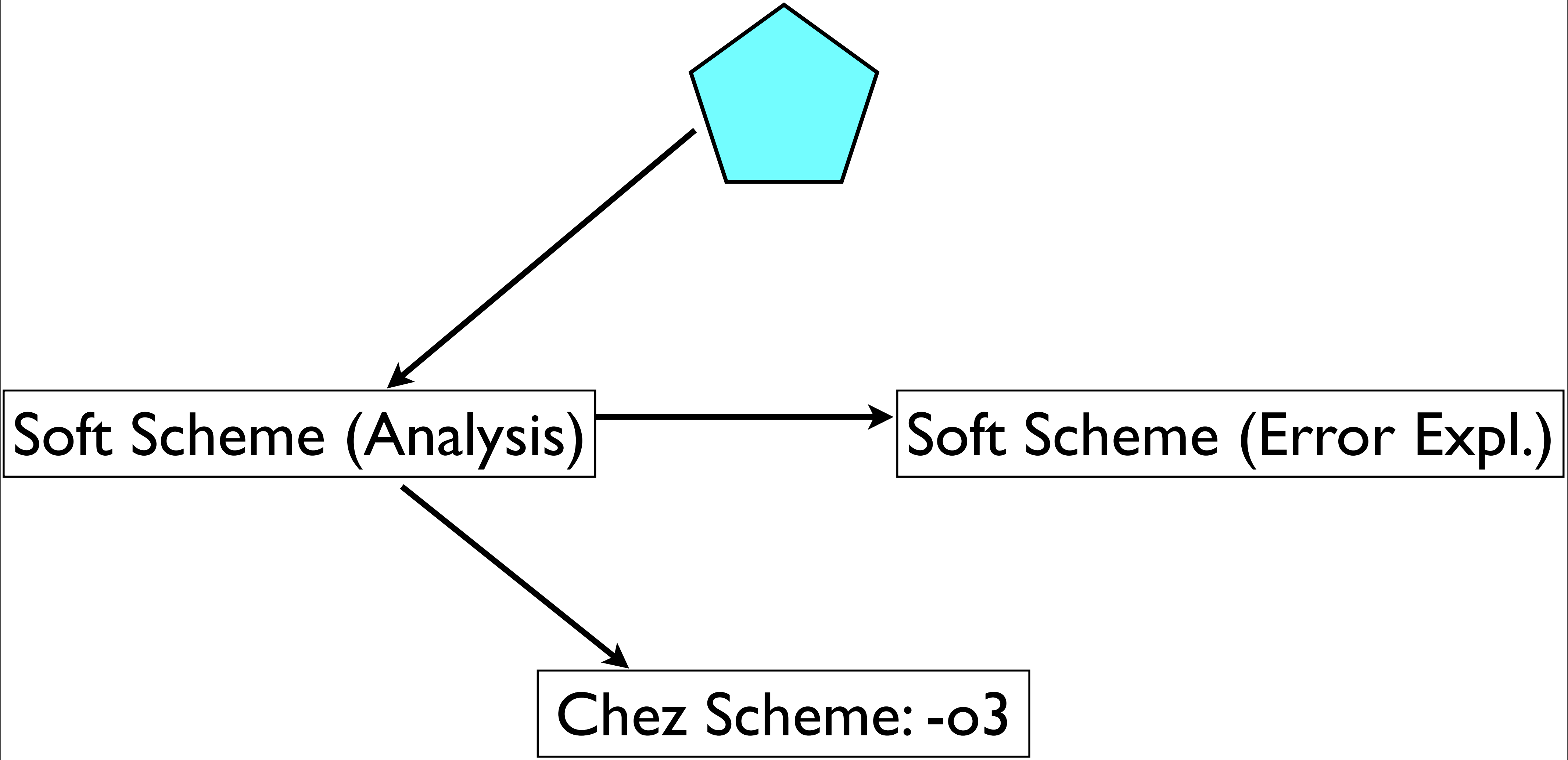
problem: Fagan can deal with nothing but toy programs

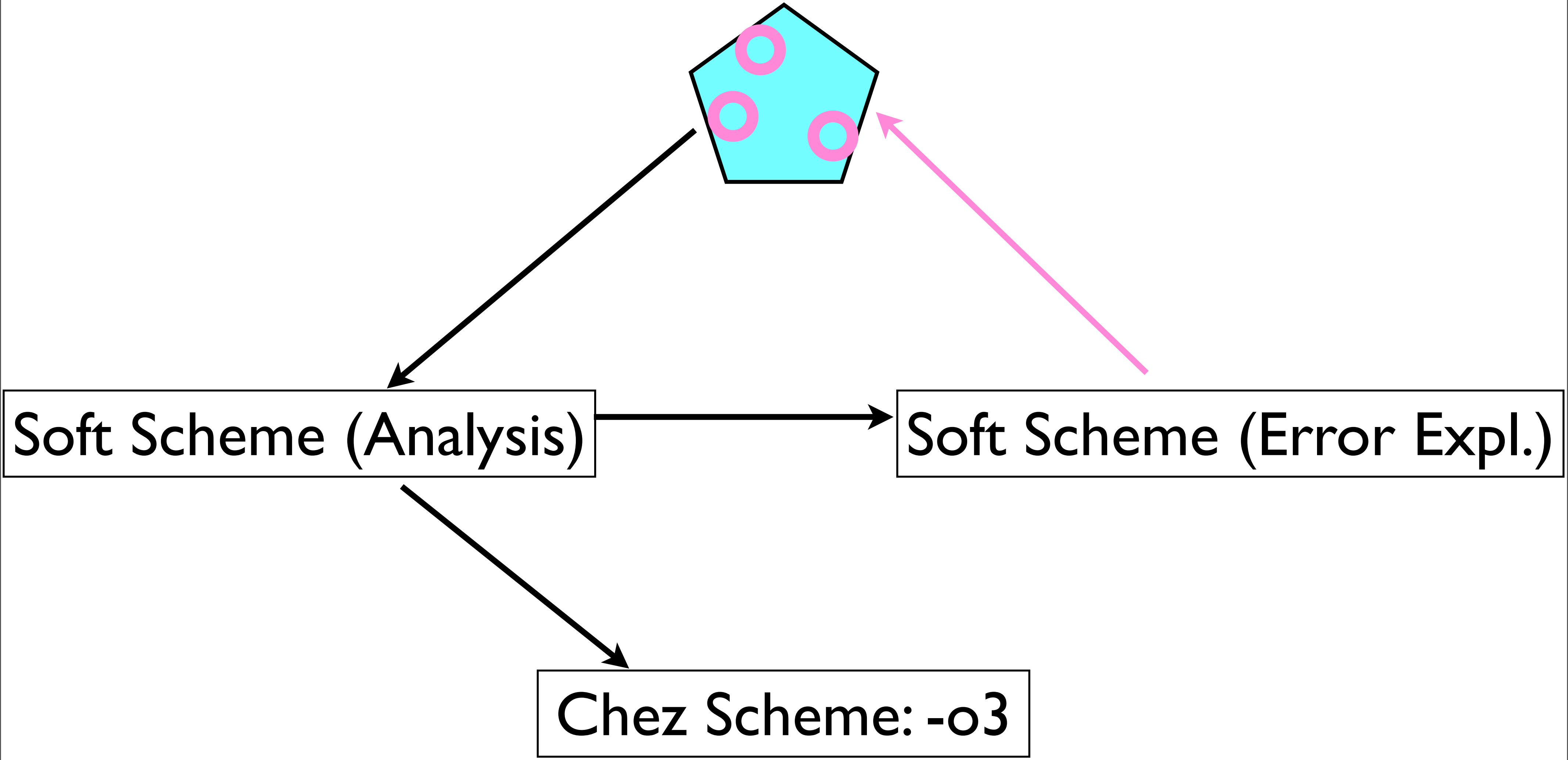
solution: improve implementation algebra;
cope with mutations, continuations, etc.

experience: absolutely, totally miserable



Soft Scheme (Analysis)
Chez Scheme: -o3


Soft Scheme (Analysis)
Soft Scheme (Error Expl.)
Chez Scheme: -o3


Soft Scheme (Analysis)
Soft Scheme (Error Expl.)
Chez Scheme: -o3

One-Year Sabbatical @ CMU:

- context: SML versus (Soft) Scheme
- many 1,000loc programs; see "Extensible Denotational Semantics" (Sendai, 1994)
- type errors in SML are difficult
- type errors in Soft Scheme are *pure torture*
- modules but no modularity with Soft Scheme

Shriram's starter project:

- context: Soft Scheme on SLaTeX
- Sitaram's SLaTeX uses every "bit" of Scheme (and Common Lisp); truly "in the wild"
- Soft Scheme discovers type problems
- explaining type errors in Soft Scheme remains for PhD-level experts
- not useful for undergraduate courses

Cormac Flanagan 1993-1998

MrSpidey

problem: Soft Scheme's error reporting; modularity

Cormac Flanagan 1993-1998

MrSpidey

problem: Soft Scheme's error reporting; modularity

solution: replace HM-style inference with
flow-based Set-Based Analysis;
tailor inference to *components*

Cormac Flanagan 1993-1998

MrSpidey

problem: Soft Scheme's error reporting; modularity

solution: replace HM-style inference with
flow-based Set-Based Analysis;
tailor inference to *components*

experience: usable with undergraduate students,
but explaining types and errors remains difficult;
performance is O(n^3) bound

```scheme
(define value
  (letrec ((get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ((x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                            (car (regexp-match "[A-Z/]+" name))))))
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
        l))))
```

$s = (t \rightarrow \underline{int})$

$t = (v \rightarrow \underline{char})$

$v = \underline{double}$

unification

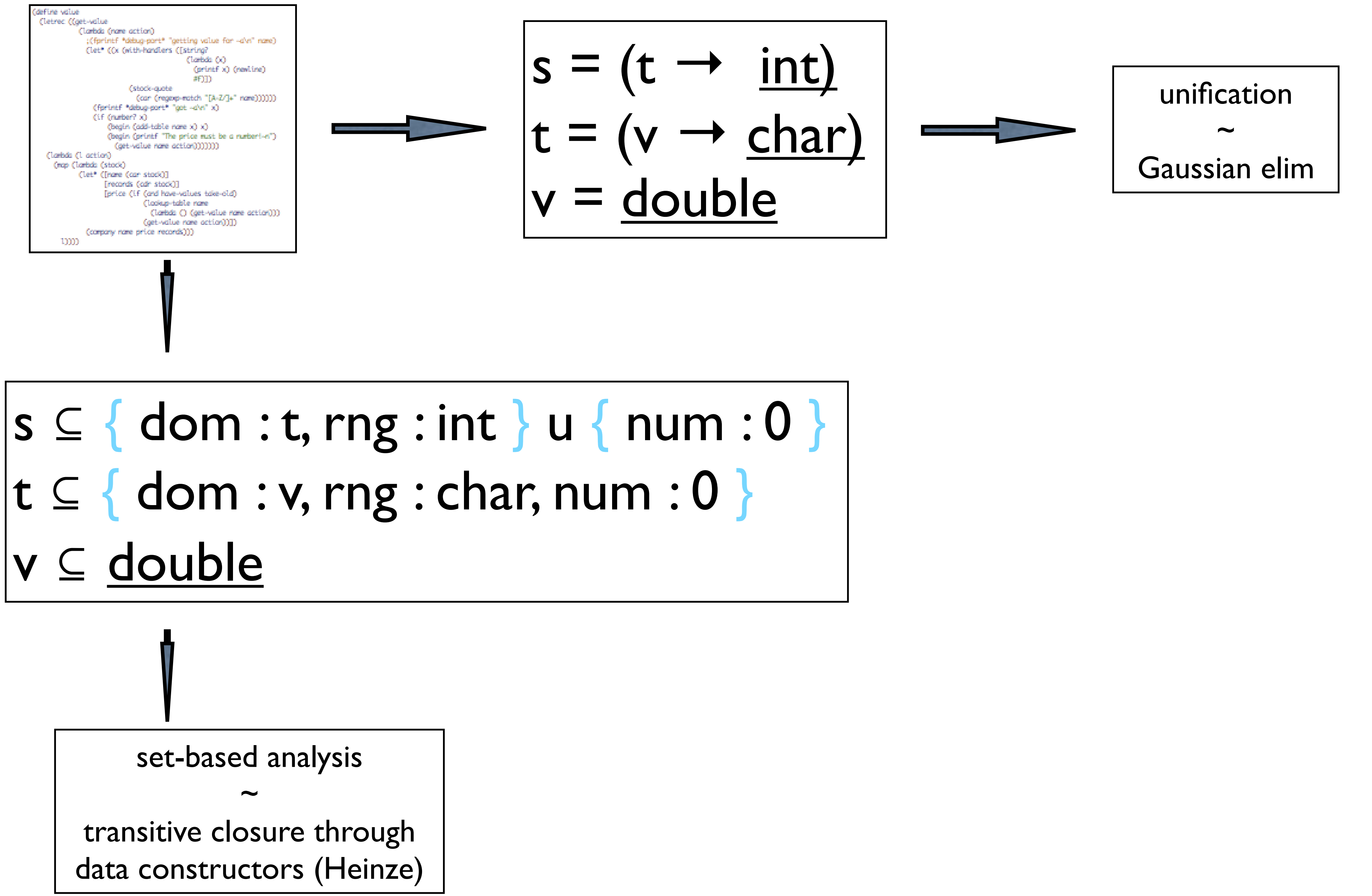
~

Gaussian elim

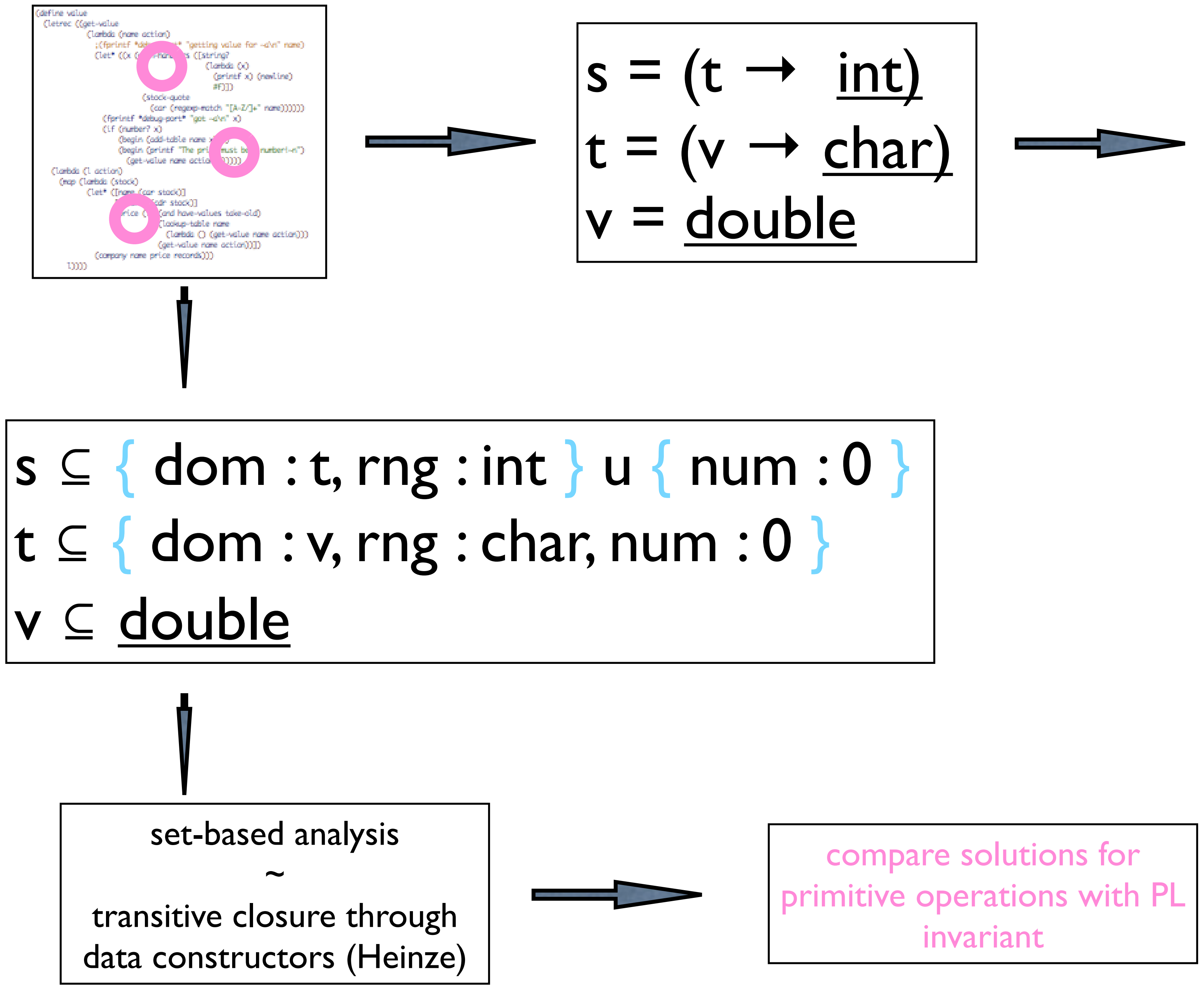
$s \subseteq \{\ dom : t, rng : int\ \} \cup \{\ num : 0\ \}$

$t \subseteq \{\ dom : v, rng : char, num : 0\ \}$

$v \subseteq \underline{double}$



(define value
(letrec ((get-value
(lambda (name action)
;(fprintf *debug-port* "getting value for ~a\n" name)
(let* ((x (with-handlers ([string?
(lambda (x)
(printf x) (newline)
#f)])
(stock-quote
(car (regexp-match "[A-Z/]+" name))))))
(fprintf *debug-port* "got ~a\n" x)
(if (number? x)
(begin (add-table name x) x)
(begin (printf "The price must be a number!~n")
(get-value name action)))))))
(lambda (l action)
(map (lambda (stock)
(let* ([name (car stock)]
[records (cdr stock)]
[price (if (and have-values take-old)
(lookup-table name
(lambda () (get-value name action)))
(get-value name action))])
(company name price records)))
l))))
s = (t → int)
t = (v → char)
v = double
unification
~
Gaussian elim
s ⊆ { dom : t, rng : int } u { num : 0 }
t ⊆ { dom : v, rng : char, num : 0 }
v ⊆ double
set-based analysis
~
transitive closure through
data constructors (Heinze)


(define value
(letrec ((get-value
(lambda (name action)
;(fprintf "getting value for ~a\n" name)
(let* ((x
([string?
(lambda (x)
(printf x) (newline)
#f)])
(stock-quote
(car (regexp-match "[A-Z/]+" name))))))
(fprintf *debug-port* "got ~a\n" x)
(if (number? x)
(begin (add-table name
(begin (printf "The pri
number!~n")
(get-value name actio
(lambda (l action)
(map (lambda (stock)
(let* ([name (car stock)]
cdr stock)]
rice (
(and have-values take-old)
(lookup-table name
(lambda () (get-value name action)))
(get-value name action))))
(company name price records)))
l))))
s = (t → int)
t = (v → char)
v = double
unification
~
Gaussian elim
s ⊆ { dom : t, rng : int } u { num : 0 }
t ⊆ { dom : v, rng : char, num : 0 }
v ⊆ double
set-based analysis
~
transitive closure through
data constructors (Heinze)
compare solutions for
primitive operations with PL
invariant

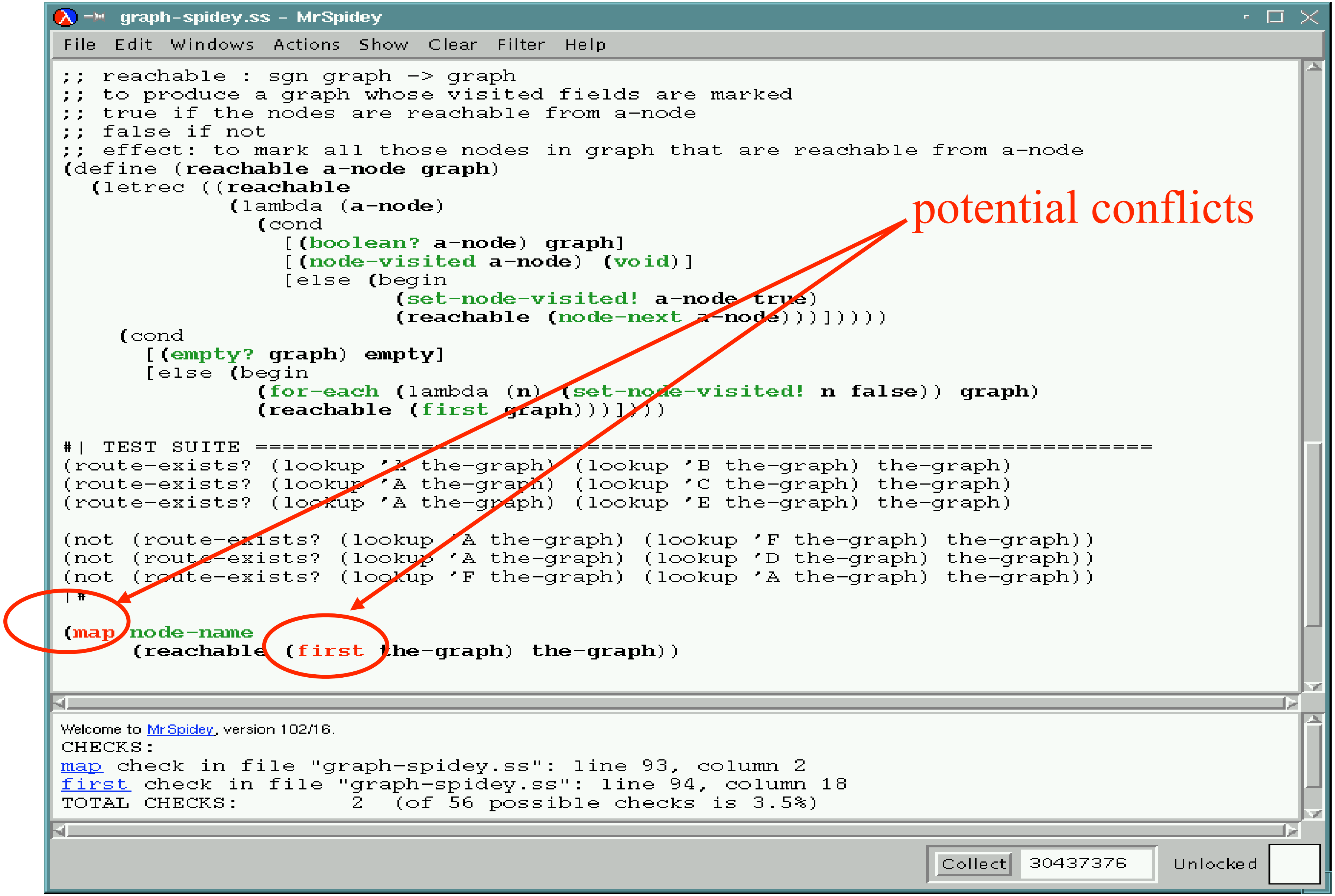
- HM performs in near-linear time in practice
- HM is easy to understand *in principle*
- HM "smears" origin information across solution due to bi-directional flow

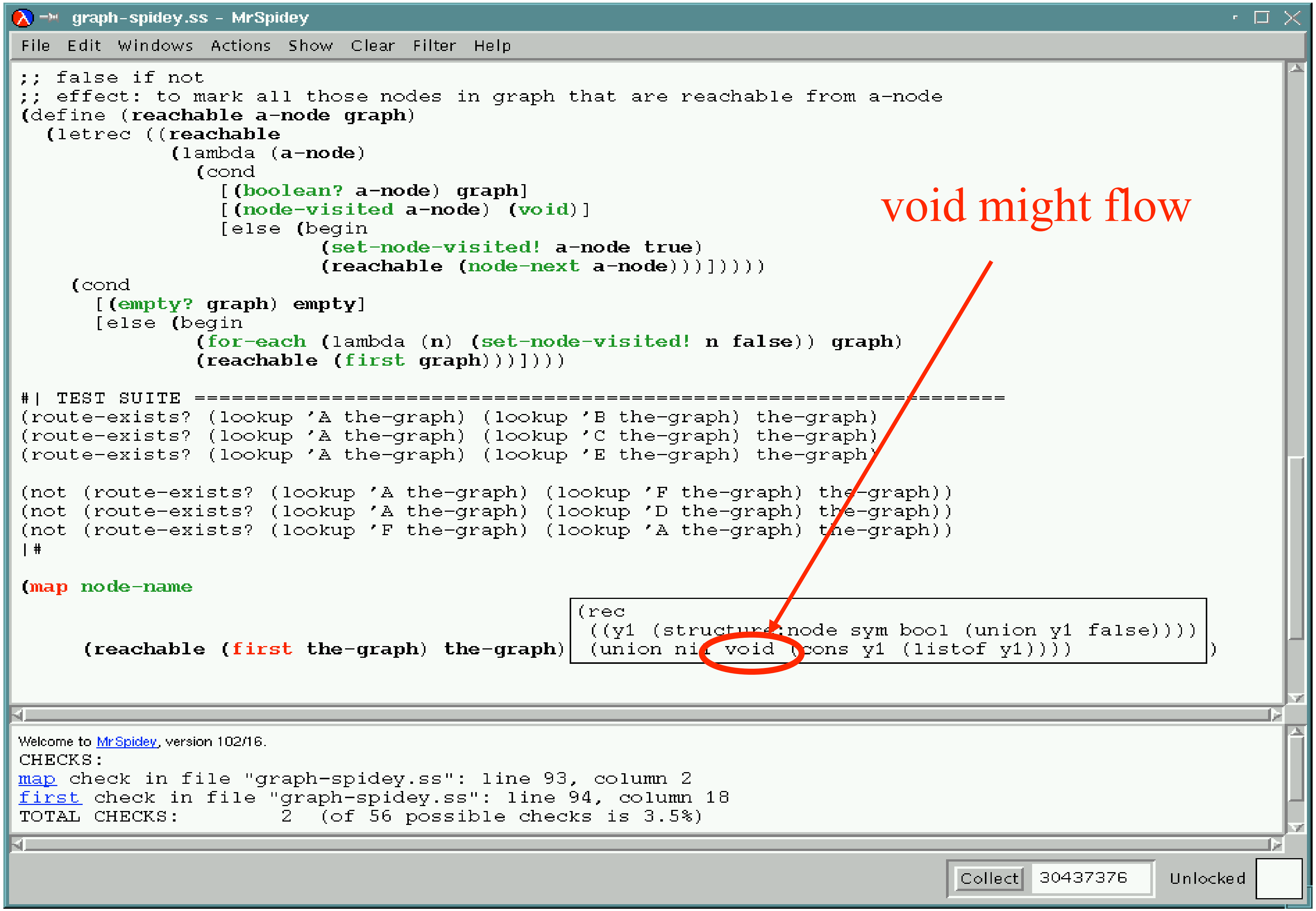
- SBA performs in linear time up to 2,500 loc
- SBA is also easy to explain to programmers
- SBA pushes information *only* along actual edges in the flow graph

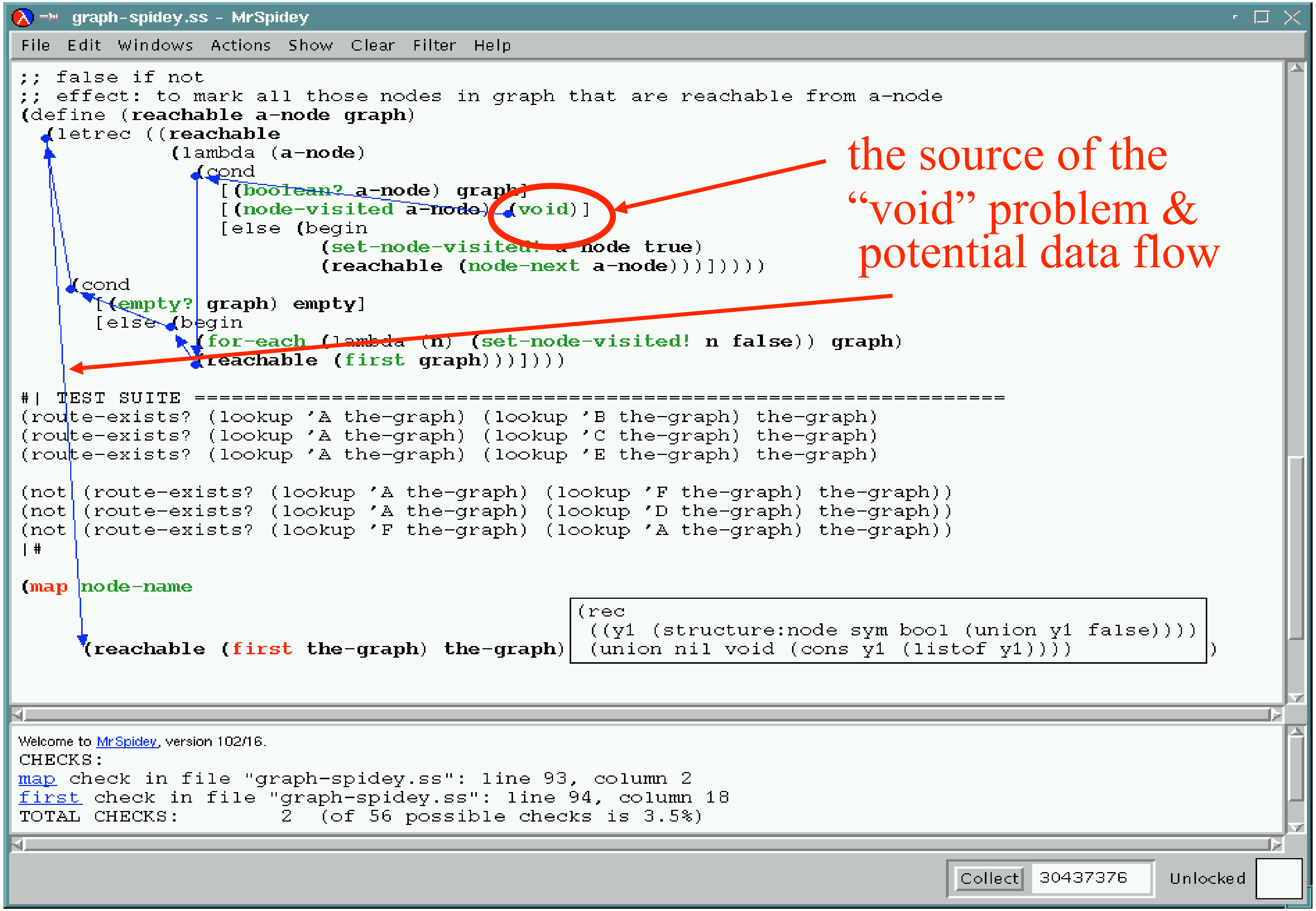
- HM performs in near-linear time in practice
- HM is easy to understand *in principle*
- HM "smears" origin information across solution due to bi-directional flow

- SBA performs in linear time up to 2,500 loc
- SBA is also easy to explain to programmers
- SBA pushes information *only* along actual edges in the flow graph

... and we can visulalize those!


graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; reachable : sgn graph -> graph
;; to produce a graph whose visited fields are marked
;; true if the nodes are reachable from a-node
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))])))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ==
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph))
potential conflicts
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376 Unlocked


graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))])))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ==
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
void might flow
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376 Unlocked


graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))]))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ==
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
)
the source of the "void" problem & potential data flow
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376 Unlocked

Components:

```
(define value
  (letrec ([get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ([x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                            (car (regexp-match "[A-Z/]+" name))))])
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))])
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
           l))))
```

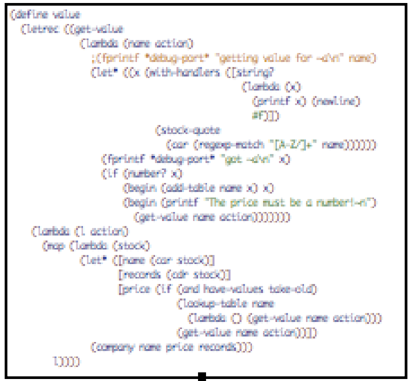
```
(define company
  (lambda (name val l)
    (let company ([list l] [base 0] [cost 0] [no-shares 0]
                  [last-price (share-price (car l))]
                  [year (year-of (car l))]
                  [yearly (cons
                            (make-year-record (- (year-of (car l)) 1)
                              (100> (price (car l)))
                              (100> (price (car l))))
                            '())])
      (cond
        [(null? list)
         (let ([current-value (<1000 (* no-shares (100> val)))])
           (make-company-record name no-shares
                                cost
                                current-value
                                (- current-value cost) ;; profit
                                base ;; tax base
                                (- current-value base) ;; capital gains
                                (cons (make-year-record year cost current-value)
                                      yearly)))]
        [(= (year-of (car list)) year)
         (let ([raw-cost (100> (price (car list)))]
               [s (1000> (shares (car list)))])
           (company (cdr list)
                    (+ base raw-cost)
                    (if (dividend? (car list)) cost (+ cost raw-cost))
                    (+ no-shares s)
                    (100> (share-price (car list)))
                    year yearly))]
        [else ;; new year
          (with-handlers ([void (lambda (e)
                                  (printf "bug ~a ~s ~s~n" name last-price no-shares)
```

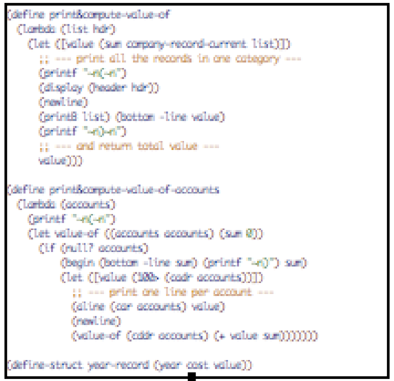
```
(define print&compute-value-of
  (lambda (list hdr)
    (let ([value (sum company-record-current list)])
      ;; --- print all the records in one category ---
      (printf "~n(~n")
      (display (header hdr))
      (newline)
      (print8 list) (bottom -line value)
      (printf "~n)~n")
      ;; --- and return total value ---
      value)))

(define print&compute-value-of-accounts
  (lambda (accounts)
    (printf "~n(~n")
    (let value-of ([accounts accounts] [sum 0])
      (if (null? accounts)
          (begin (bottom -line sum) (printf "~n)") sum)
          (let ([value (100> (cadr accounts))])
            ;; --- print one line per account ---
            (aline (car accounts) value)
            (newline)
            (value-of (cddr accounts) (+ value sum)))))))

(define-struct year-record (year cost value))
```

Components:





Constraints:

$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq \underline{double}$

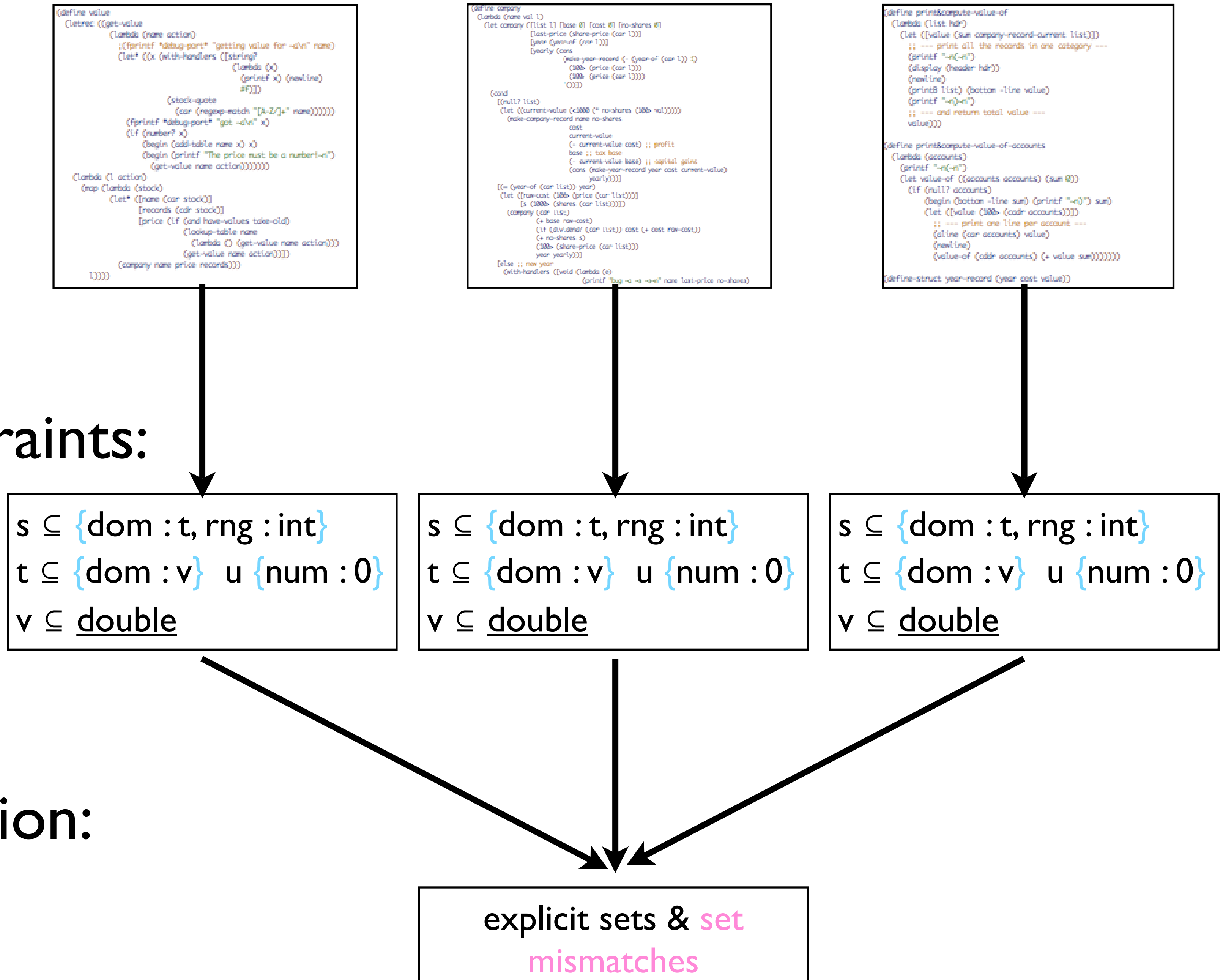
$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq \underline{double}$

$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq \underline{double}$

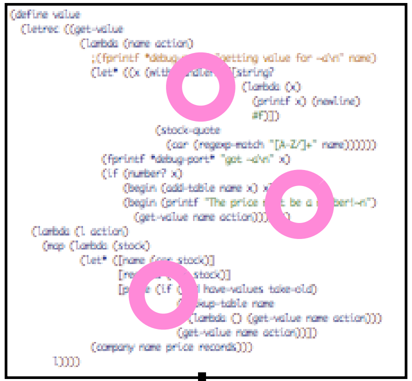
Components:

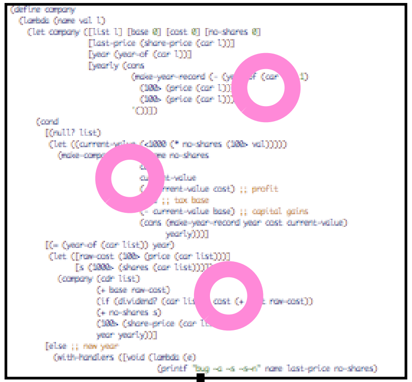
Constraints:

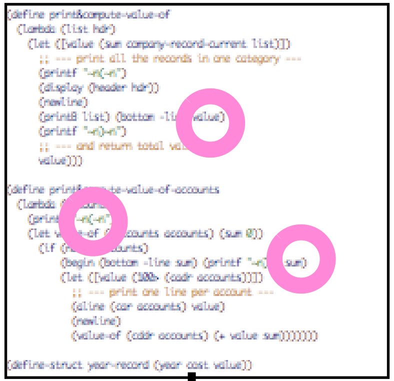
Solution:



Components:







Constraints:

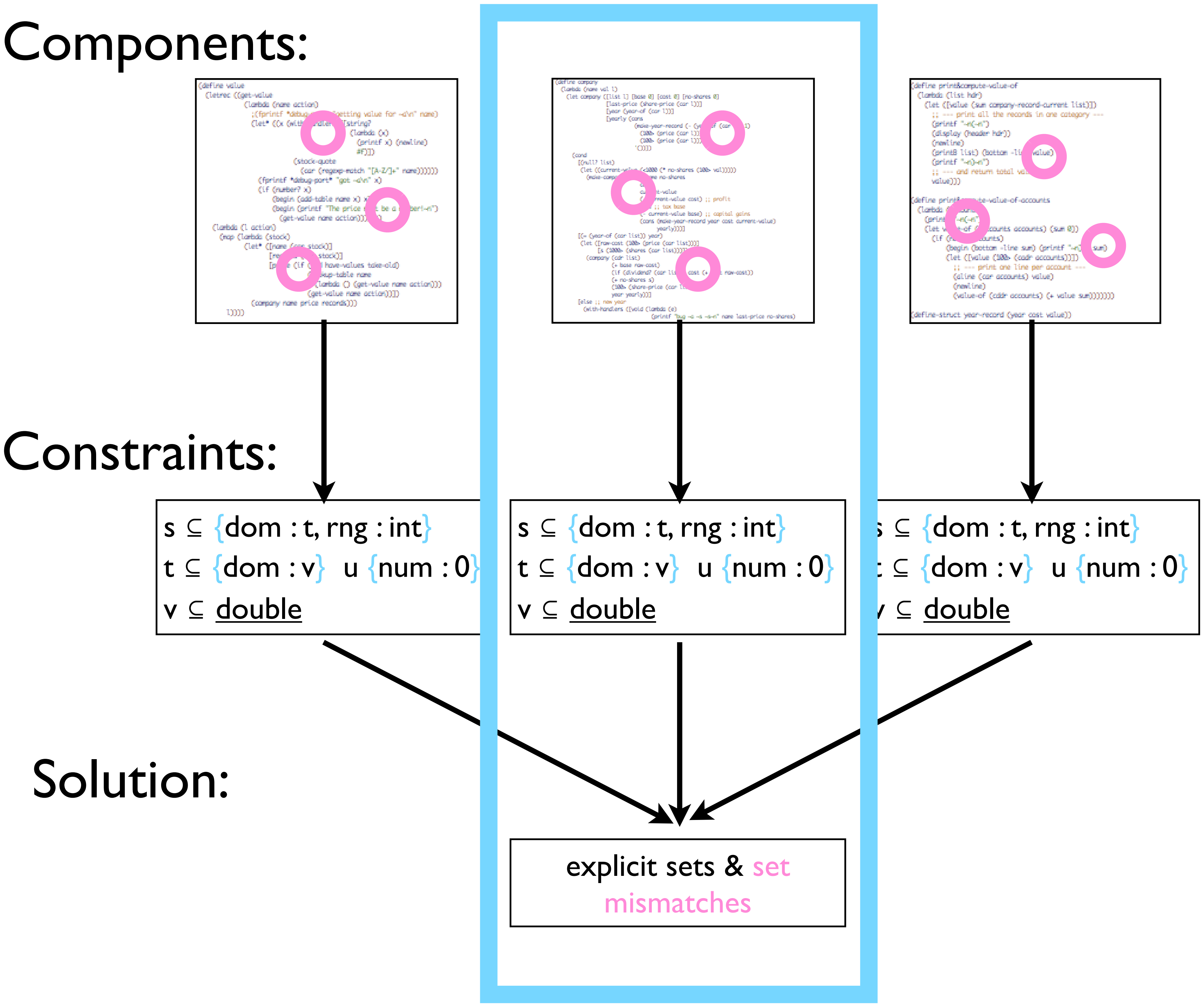
$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq$ double

$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq$ double

$s \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\} \cup \{num : 0\}$
$v \subseteq$ double

Solution:

explicit sets & set mismatches


Components:
Constraints:
s ⊆ {dom : t, rng : int}
t ⊆ {dom : v} ∪ {num : 0}
v ⊆ double
s ⊆ {dom : t, rng : int}
t ⊆ {dom : v} ∪ {num : 0}
v ⊆ double
s ⊆ {dom : t, rng : int}
t ⊆ {dom : v} ∪ {num : 0}
v ⊆ double
Solution:
explicit sets & set mismatches

- creating and storing constraint sets: quadratic only over a certain size
- re-computing the solution from just one set is cheaper than computing it from all
- *... but can't add add untyped modules or treat existing component as modules in untyped code*

- creating and storing constraint sets: quadratic only over a certain size
- re-computing the solution from just one set is cheaper than computing it from all
- *... but can't add add untyped modules or treat existing component as modules in untyped code*

In sum, it isn't *really* modular.

Experience:

- In 2 years: from 2,000 loc to 50,000 loc
- Personal: dozens of programs, including use as a refactoring tool
- MrSpidey finds more mistakes than Soft Scheme and most of Soft Scheme's "casts"
- Used in two undergraduate courses with some success
- Shriram continues the SLaTeX experiment

The key obstacle for Soft Scheme *and* MrSpidey:

the brittle nature of type inference

```
;; Nat -> ...
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    ... )
```

```
;; somewhere else in the program:
      ...  some-variable ...
```

```
;; Nat -> ...
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    ... )
```

```
;; somewhere else in the program:
      ...  some-variable ...
```

```
(union #f  Nat)
```

```
;; Nat -> ...
(define (dispatch-table n)
  (let ([v (make-vector n)])
    ;; --- vector set up code
    (let loop ([i 0])
      (unless (>= i n)
        (vector-set! v i (lambda (x) ...))
        (loop (+ i 1))))
    ;; --- client code
    ... )
```

```
;; somewhere else in the program:
      ...  some-variable ...
```

```
;; Nat -> ...
(define (dispatch-table n)
  (let ([v (make-vector n)])
    ;; --- vector set up code
    (let loop ([i 0])
      (unless (>= i n)
        (vector-set! v i (lambda (x) ...))
        (loop (+ i 1))))
    ;; --- client code
    ... )
```

```
;; somewhere else in the program:
      ...  some-variable ...
```

```
(union #f
           ...
           ...
           ... ;; some 20 lines
           ...  )
```

```
;; Nat -> ...
(define (dispatch-table n)
  (let ([v (make-vector n)])
    ;; --- vector set up code
    (let loop ([i 0])
      (unless (>= i n)
        (vector-set! v i (lambda (x) ...))
        (loop (+ i 1))))
    ;; --- client code
    ... )
```

```
;; somewhere else in the program:
      ...  some-variable ...
```

Small syntactic changes without semantic meaning imply large changes to inferred types

```
(union #f
           ...
           ...
           ... ;; some 20 lines
           ...  )
```

Philippe Meunier 1999-2006

problem: MrSpidey's curious imprecision; modularity again

Philippe Meunier 1999-2006

problem: MrSpidey's curious imprecision; modularity again

solution: introduce explicit types,
with occurrence typing for functional Scheme

Philippe Meunier 1999-2006

problem: MrSpidey's curious imprecision; modularity again

solution: introduce explicit types,
with occurrence typing for functional Scheme

solution: introduce explicit contracts for
Scheme modules and combine with SBA-style
inference

Philippe Meunier 1999-2006

problem: MrSpidey's curious imprecision; modularity again

solution: introduce explicit types,
with occurrence typing for functional Scheme

solution: introduce explicit contracts for Scheme modules and combine with SBA-style inference

experience: prototypes only

Sam Tobin-Hochstadt 2005-2009

From Scripts to Programs

problem: MrSpidey and friends infer brittle and large types; errors remain difficult to explain and fix

Sam Tobin-Hochstadt 2005-2009

From Scripts to Programs

problem: MrSpidey and friends infer brittle and large types; errors remain difficult to explain and fix

solution: replace inference with explicit static types; support sound and incremental approach to type enrichment with contracts;

Sam Tobin-Hochstadt 2005-2009

From Scripts to Programs

problem: MrSpidey and friends infer brittle and large types; errors remain difficult to explain and fix

solution: replace inference with explicit static types; support sound and incremental approach to type enrichment with contracts;

with influence from Gray, Findler, & Flatt, "Fine-Grained Interoperability through Contracts and Mirrors" (OOPSLA '05)

Sam Tobin-Hochstadt 2005-2009

From Scripts to Programs

problem: MrSpidey and friends infer brittle and large types; errors remain difficult to explain and fix

solution: replace inference with explicit static types; support sound and incremental approach to type enrichment with contracts;

with influence from Gray, Findler, & Flatt, “Fine-Grained Interoperability through Contracts and Mirrors” (OOPSLA ‘05)

experience: usable with undergraduate students, but remains extremely difficult

Incremental Type Enrichment

Soft (HM, SBA) Typing

- type *all* untyped, complete programs
- use casts to bridge problems and inform programmer
- allow programmers to debug type problems with the dynamic debugger

Typed Scheme

- type *some* untyped programs; *fail others*
- *incrementally* enrich untyped programs with types
- synthesize *contracts* to ensure *soundness* for mixed typed and untyped programs

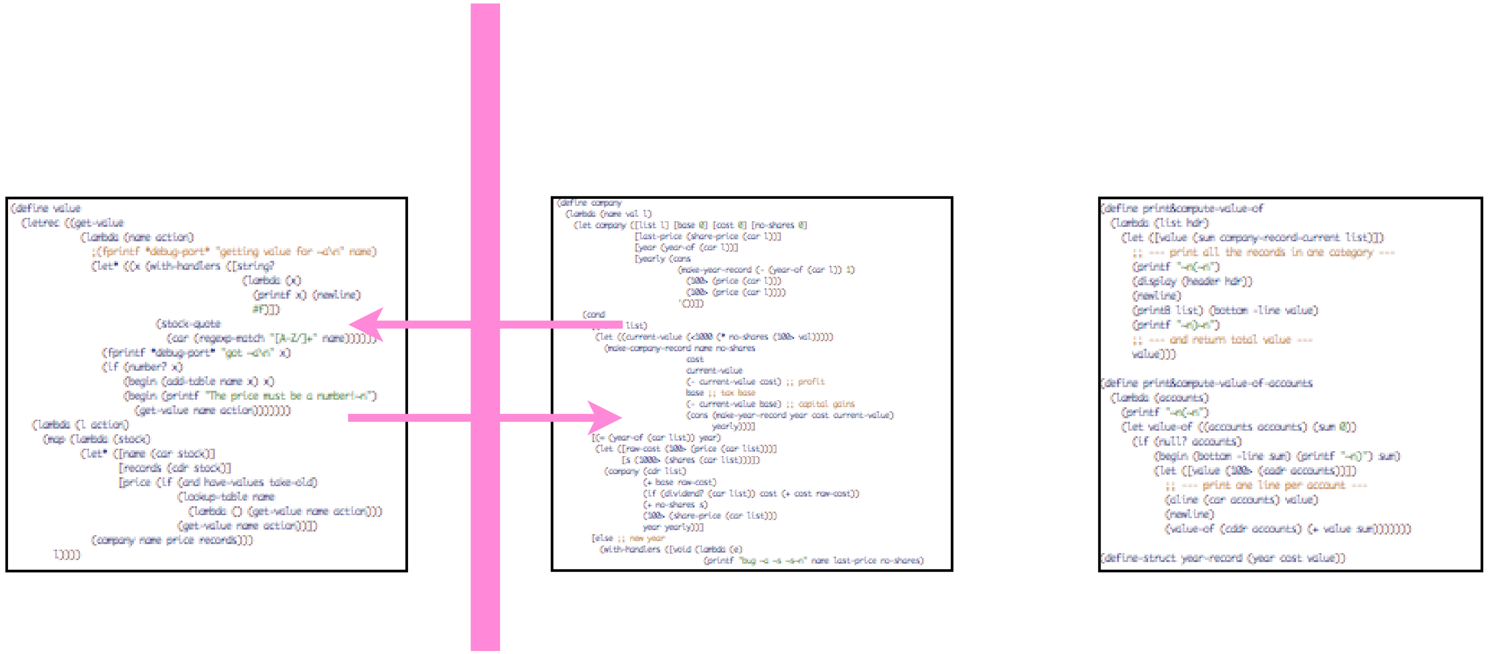
Why?

- Motivation I: work on web programming in late 1990s and early 2000
- Motivation II: Meunier's failure to cope with explicit types and implicit subtyping

How?

- Step I: Findler and Felleisen on contracts for higher-order languages
- Step II: incremental conversion, soundness, and blame
- Step III: design and validation of a practical type system

Step 1: Contracts: Types for Scheme and Beyond

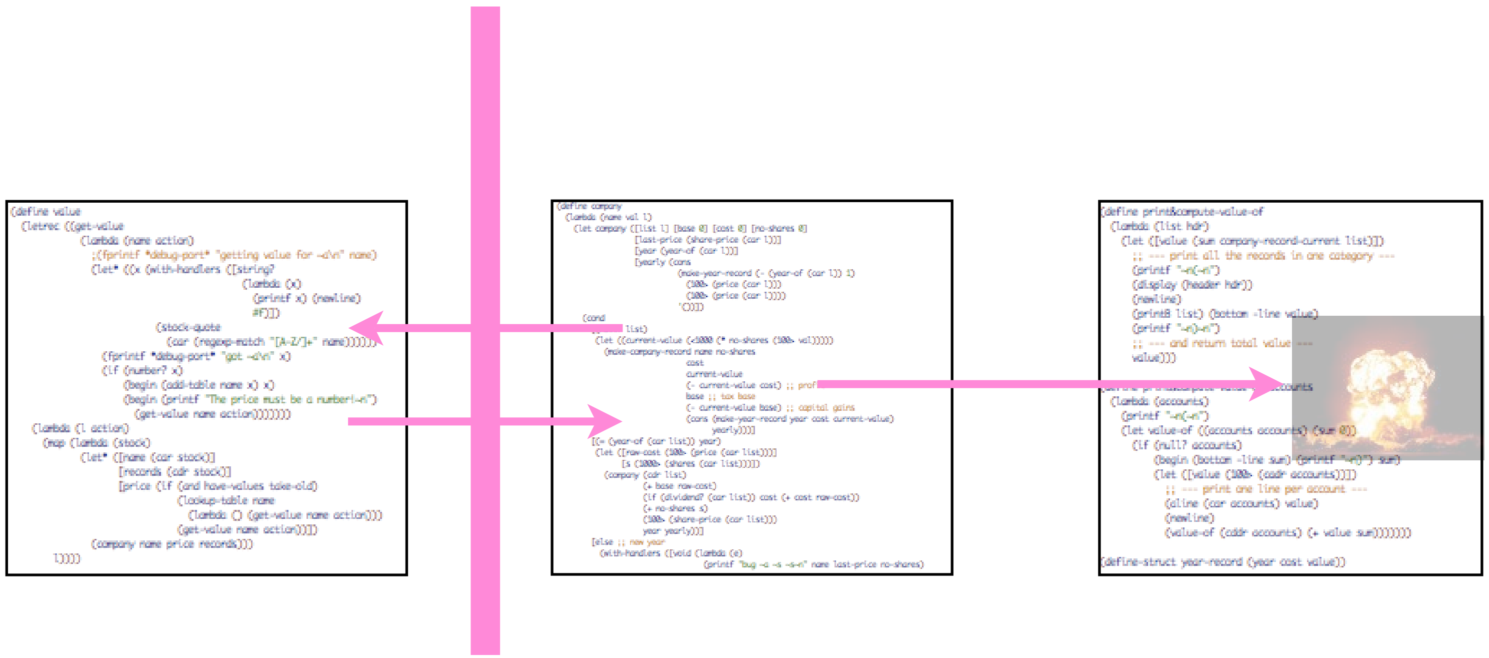


[e : (integer? ---> (and/c natural-number/c **prime?**))] [s : string?]

---d--->

(and/c string? **(lambda (r) (string=? (decode e r) s)))**

Step 1: Contracts: Types for Scheme and Beyond

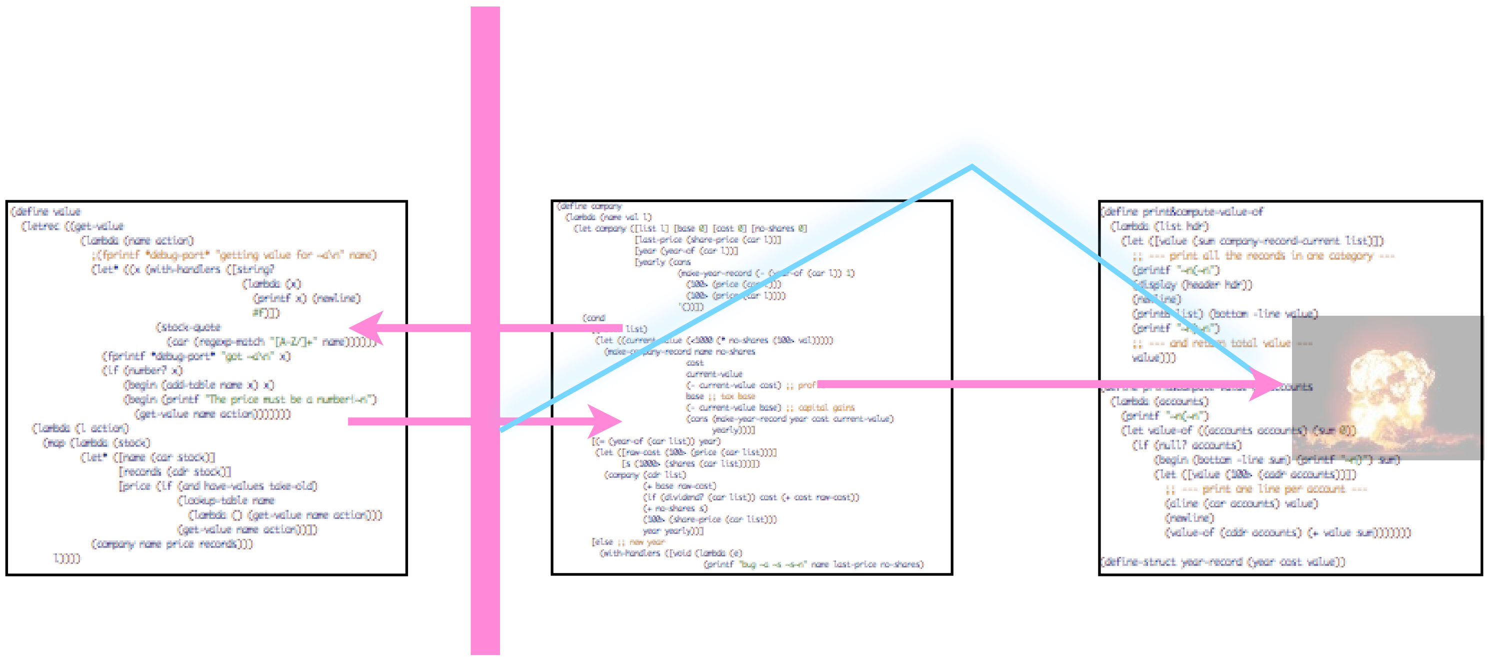


[e : (integer? ---> (and/c natural-number/c **prime?**))] [s : string?]

---d--->

(and/c string? **(lambda (r) (string=? (decode e r) s)))**

Step 1: Contracts: Types for Scheme and Beyond

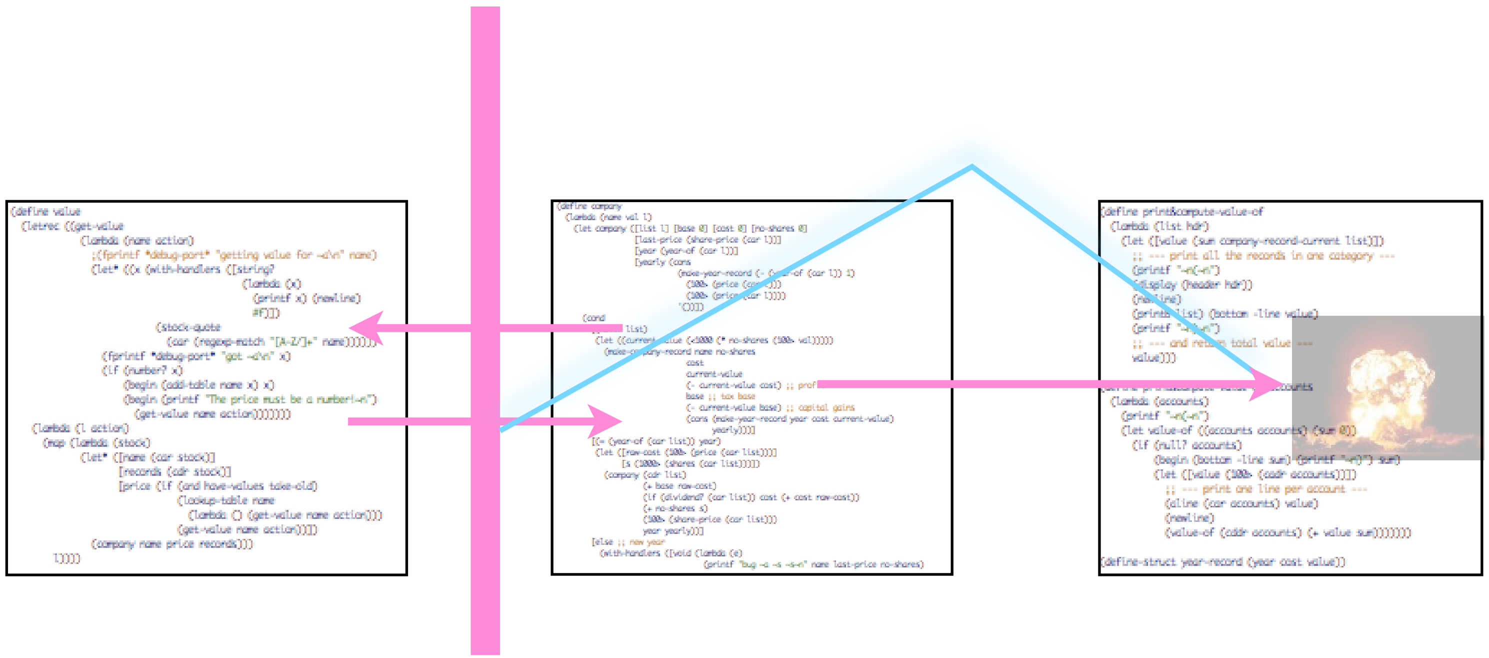


[e : (integer? ---> (and/c natural-number/c **prime?**))] [s : string?]

---d--->

(and/c string? **(lambda (r) (string=? (decode e r) s)))**

Step 1: Contracts: Types for Scheme and Beyond



[e : (integer? ---> (and/c natural-number/c **prime?**))] [s : string?]

---d--->

(and/c string? **(lambda (r) (string=? (decode e r) s)))**

note: type-like contracts are (higher-order) casts

Step II: The Research Framework

```scheme
(define value
  (letrec ([get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ([x (with-handlers ([string?
                                         (lambda (x)
                                           (printf x) (newline)
                                           #f)])
                          (stock-quote
                            (car (regexp-match "[A-Z/]+" name))))])
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))])
    (lambda (l action)
      (map (lambda (stock)
            (let* ([name (car stock)]
                   [records (cdr stock)]
                   [price (if (and have-values take-old)
                              (lookup-table name
                                (lambda () (get-value name action)))
                              (get-value name action))])
              (company name price records)))
        l))))
```

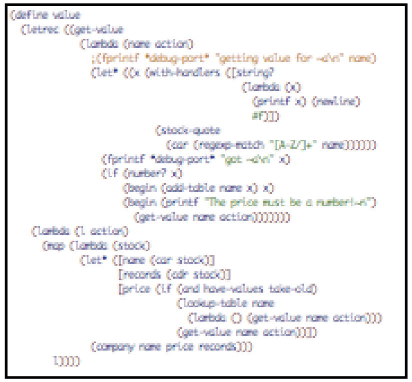
```scheme
(define company
  (lambda (name val l)
    (let company ([list l] [base 0] [cost 0] [no-shares 0]
                  [last-price (share-price (car l))]
                  [year (year-of (car l))]
                  [yearly (cons
                            (make-year-record (- (year-of (car l)) 1)
                              (100> (price (car l)))
                              (100> (price (car l))))
                            '())])
      (cond
        [(null? list)
         (let ([current-value (<1000 (* no-shares (100> val)))])
           (make-company-record name no-shares
                                cost
                                current-value
                                (- current-value cost) ;; profit
                                base ;; tax base
                                (- current-value base) ;; capital gains
                                (cons (make-year-record year cost current-value)
                                      yearly)))]
        [(= (year-of (car list)) year)
         (let ([raw-cost (100> (price (car list)))]
               [s (1000> (shares (car list)))])
           (company (cdr list)
                    (+ base raw-cost)
                    (if (dividend? (car list)) cost (+ cost raw-cost))
                    (+ no-shares s)
                    (100> (share-price (car list)))
                    year yearly))]
        [else ;; new year
         (with-handlers ([void (lambda (e)
                                 (printf "bug ~a ~s ~s~n" name last-price no-shares)
```

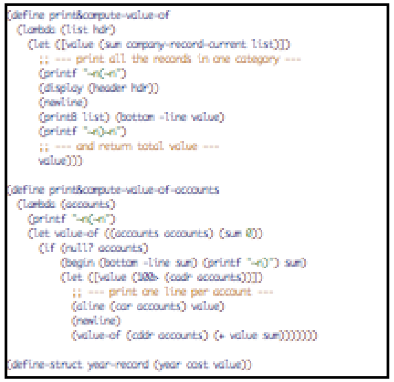
```scheme
(define print&compute-value-of
  (lambda (list hdr)
    (let ([value (sum company-record-current list)])
      ;; --- print all the records in one category ---
      (printf "~n(~n")
      (display (header hdr))
      (newline)
      (print8 list) (bottom -line value)
      (printf "~n)~n")
      ;; --- and return total value ---
      value)))

(define print&compute-value-of-accounts
  (lambda (accounts)
    (printf "~n(~n")
    (let value-of ((accounts accounts) (sum 0))
      (if (null? accounts)
          (begin (bottom -line sum) (printf "~n)") sum)
          (let ([value (100> (cadr accounts))])
            ;; --- print one line per account ---
            (aline (car accounts) value)
            (newline)
            (value-of (cddr accounts) (+ value sum)))))))

(define-struct year-record (year cost value))
```

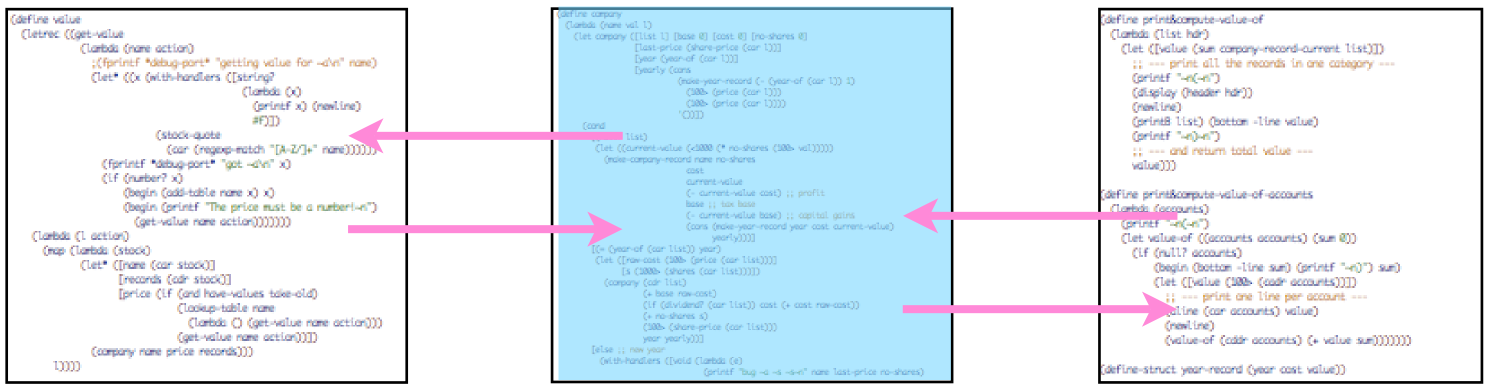
Step II: The Research Framework





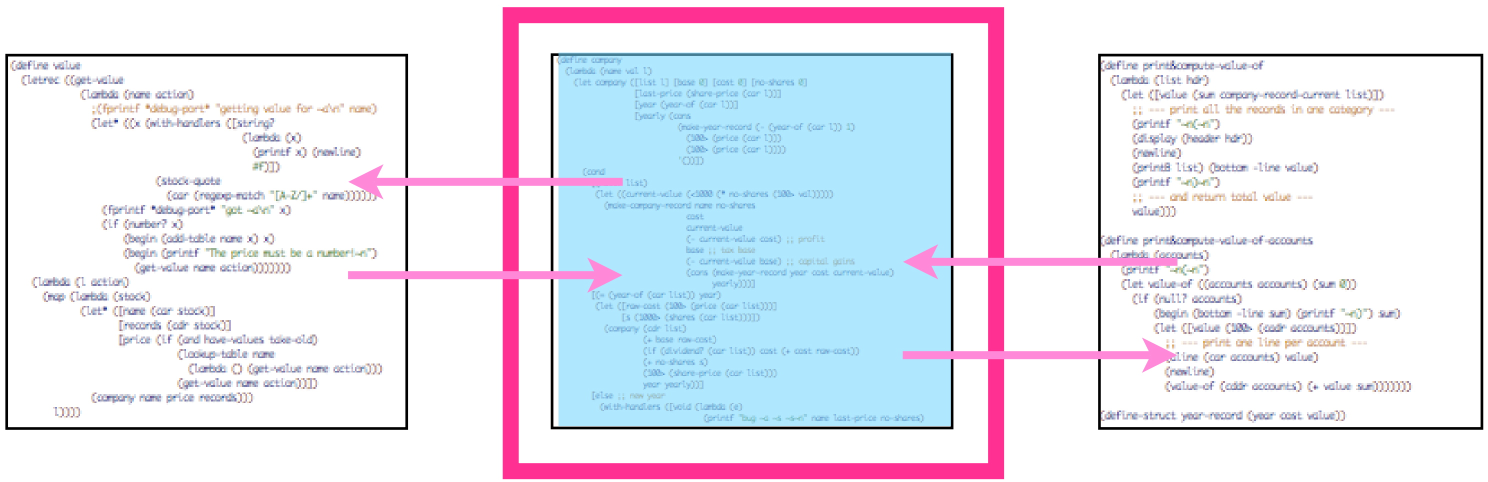
equip with types
in *sound* manner
and *identify violators*

Step II: The Research Framework



equip with types
in *sound* manner
and *identify violators*

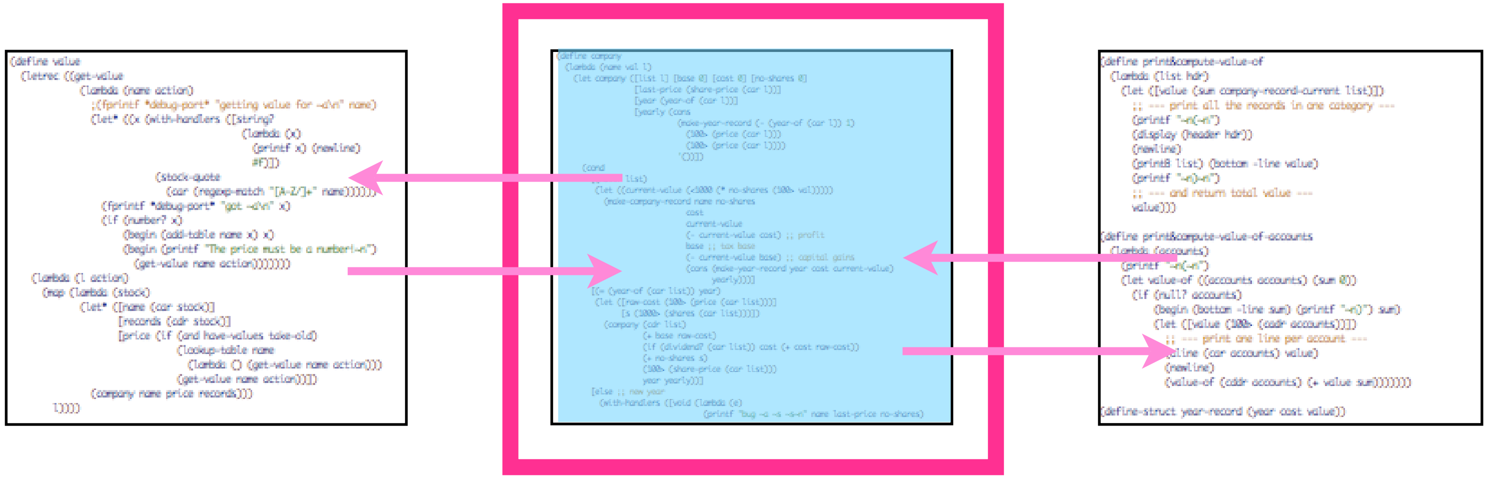
Step II: The Research Framework



equip with types
in *sound* manner
and *identify violators*

synthesize contracts (casts)
from type specs of
exports & imports

Step II: The Research Framework



equip with types
in *sound* manner
and *identify violators*

synthesize contracts (casts)
from type specs of
exports & imports

The original blame calculus and theorem,
Tobin-Hochstadt & Felleisen, DLS/OOPSLA '06

Step III: From Theory to Practice, From Scripts to Programs

From “soft” types
to “hard” types

- subtyping
- subtyping from control flow (“if splitting”)
- “true” unions
- tables, records, accessors
- polymorphism

Coping with all of
PLT Scheme

- paths (caddadr)
- variable-arity functions and multiple values
- apply
- macros
- classes, mixins, traits, unit/ components, ... continuations

Step III: Don’t Change More than You Absolutely Must!

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (circ-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

Step III: Don't Change More than You Absolutely Must!

accommodate PLT Scheme programing idioms
as they occur "in the wild" (3% rule)

```
(define-struct: rect ((nw :Any) (width : Number) (height : Number)))
(define-struct: circ ((cntr :Any)  (radius : Number)))
(define-struct: over ((top : Shape) (bot : Shape)))

(define-type-alias Shape (Rec Shape (U Plain over [Listof Plain])))
(define-type-alias Plain (U rect circ))
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
(: area (Shape -> Number))
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
(: plain? (Any -> Boolean : Plain))
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

(: plain-area (Plain -> Number))
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (circ-area s)]))

(: rect-area (rect -> Number))
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

Step III: Don't Change More than You Absolutely Must!

Step III: Don’t Change More than You Absolutely Must!

```
;; LSN = '()  | (cons Number LSN) |  (cons Symbol LSN)

;; LSN -> Number
;; add all numbers in this lsn
(define (sum lsn)
  (cond
    [(null? lsn) 0]
    [(number? (car lsn)) (+ (car lsn) (sum (cdr lsn)))]
    [else (sum (cdr lsn))]))
```

Step III: Don't Change More than You Absolutely Must!

```
;; LSN = '()  | (cons Number LSN) |  (cons Symbol LSN)

;; LSN -> Number
;; add all numbers in this lsn
(define (sum lsn)
  (cond
    [(null? lsn) 0]
    [(number? (car lsn)) (+ (car lsn) (sum (cdr lsn)))]
    [else (sum (cdr lsn))]))
```

Step III: Don't Change More than You Absolutely Must!

```
;; LSN = '()  | (cons Number LSN) |  (cons Symbol LSN)

;; LSN -> Number
;; add all numbers in this lsn
(define (sum lsn)
  (cond
    [(null? lsn) 0]
    [(number? (car lsn)) (+ (car lsn) (sum (cdr lsn)))]
    [else (sum (cdr lsn))]))
```

```
(define-type-alias LSN (U  '()  | (cons Number LSN) |  (cons Symbol LSN))

(: sum (LSN -> Number))
;; add all numbers in this lsn
(define (sum lsn)
  (cond
    [(null? lsn) 0]
    [else (let ([fst (car lsn)])
            (cond
              [(number? fst)  (+ fst (sum (cdr lsn)))]
              [else (sum (cdr lsn))]))]))
```

Experience:

- formative eval: ~5,000 loc from books, base
- summative eval: ~30,000 loc incl. code base
- ~20 volunteers have created interfaces for libs
- undergraduates: comfortable going from Scheme to Typed Scheme (lift in class gpa!)
- we are on our way to the 3% level and below for core (mostly functional) PLT Scheme

Stevie Strickland 2008-2011

Typed **PLT** Scheme

problem: PLT Scheme comes with classes,
mixins, traits, and units;
also missing: nested and arbitrary contract boundaries

Stevie Strickland 2008-2011

Typed **PLT** Scheme

problem: PLT Scheme comes with classes, mixins, traits, and units; also missing: nested and arbitrary contract boundaries

solution: theory, practice, and evaluation
exploit more of Kathy Gray's recent work

Stevie Strickland 2008-2011

Typed **PLT** Scheme

problem: PLT Scheme comes with classes, mixins, traits, and units;
also missing: nested and arbitrary contract boundaries

solution: theory, practice, and evaluation
exploit more of Kathy Gray's recent work

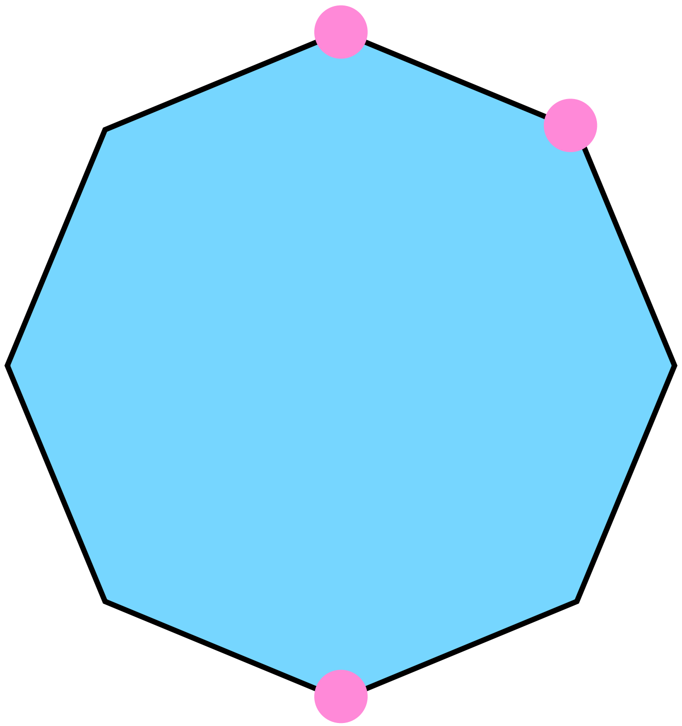
experience: hopefully some future ECOOP or OOPSLA

Lessons Learned

Lessons Learned

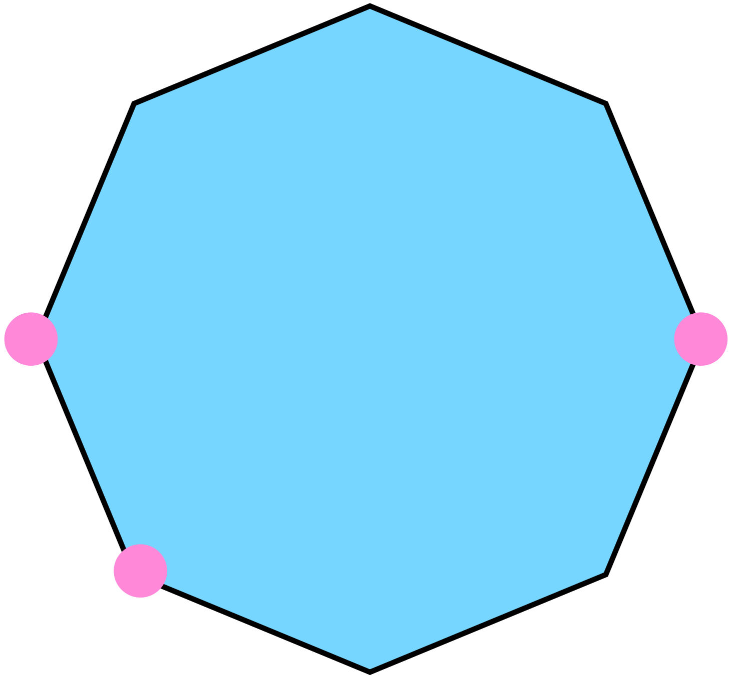
- a research agenda of moving from untyped to typed programs for 20 years
- ... from rapid prototyping to stable programs
- ... from untyped to understandable code
- ... from scripts to programs (no need for statistics)

Lessons Learned



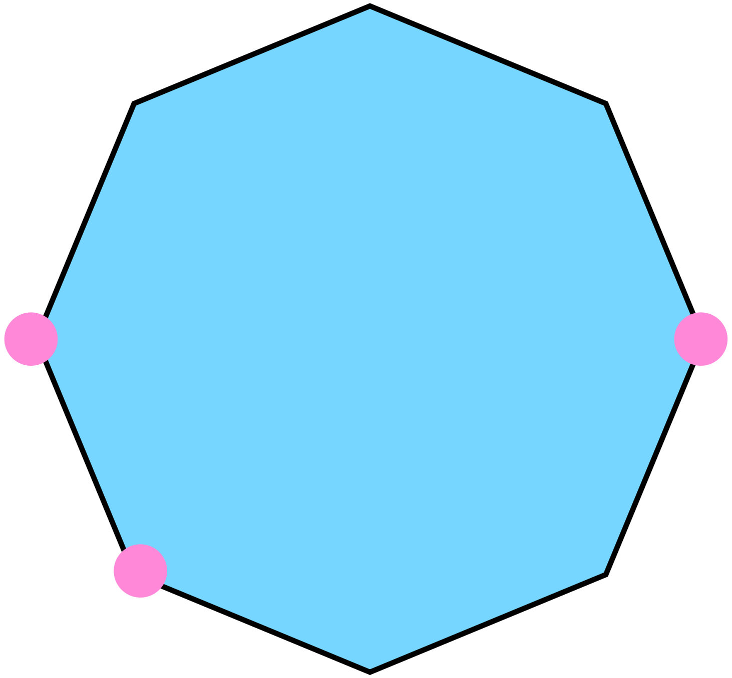
- the goal
- the PL (subject)
- granularity of incremental conversion steps?
- explicit types and the role of type inference
- the necessary quality level

The Goal



- is it about bug finding?
- do we care about soundness?

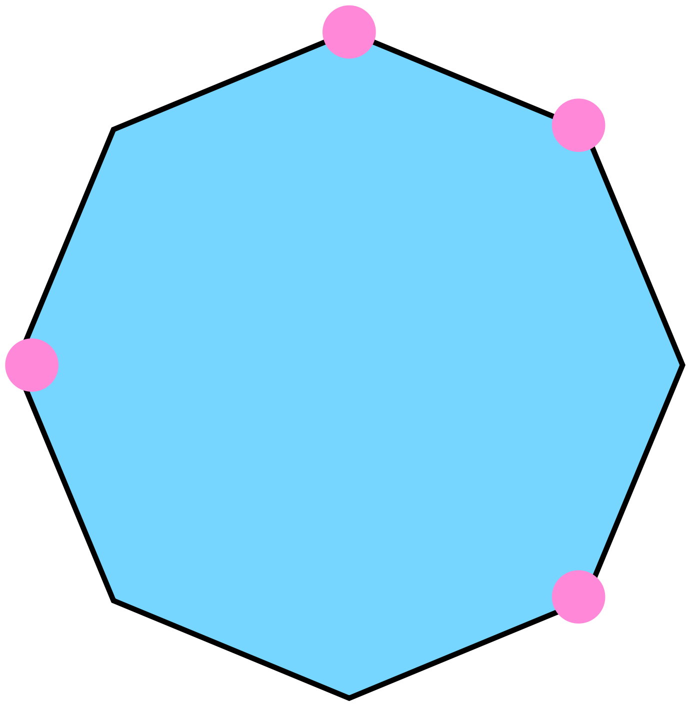
The Goal



- is it about bug finding?
- do we care about soundness?

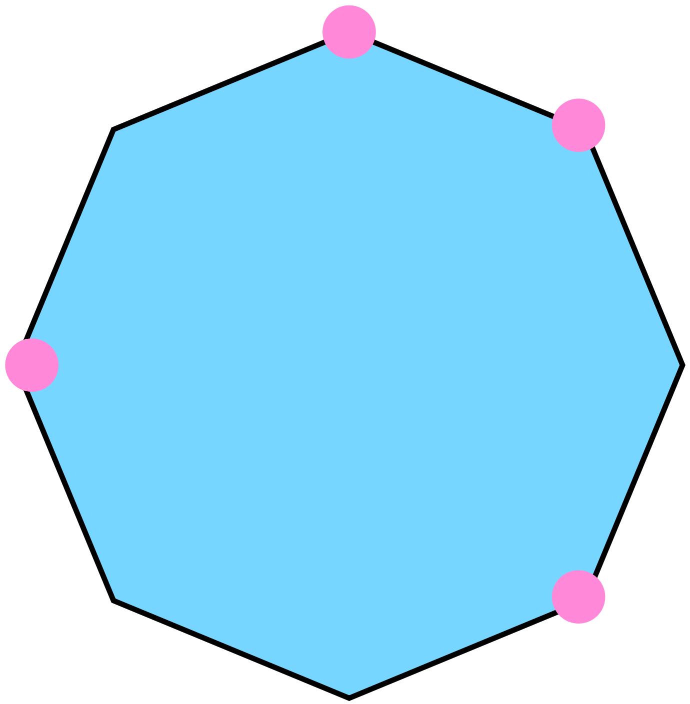
for us: adding explicit design information and facilitating future maintenance; if academics don't worry about soundness, nobody will (and see where that got us)

The Programming Language



- model (LC, Obj) vs real
- existing vs newly designed
- industrial vs academic

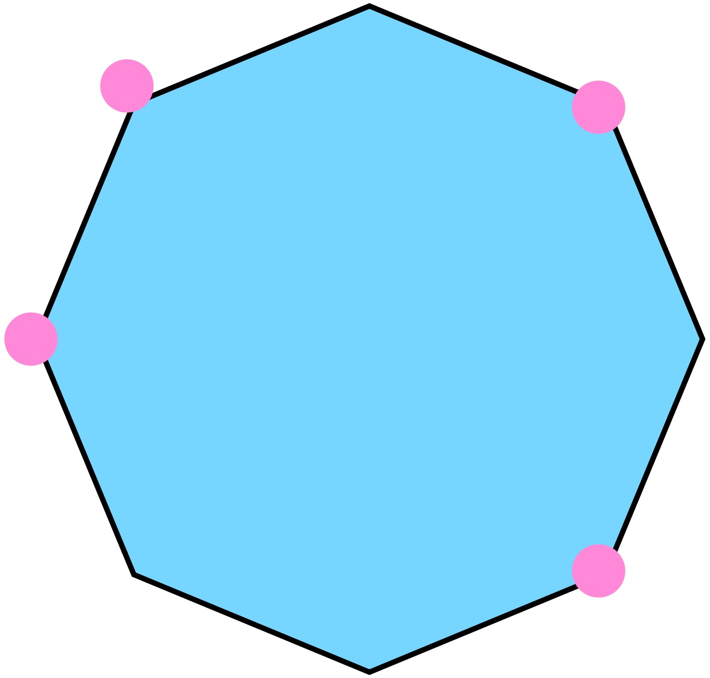
The Programming Language



- model (LC, Obj) vs real
- existing vs newly designed
- industrial vs academic

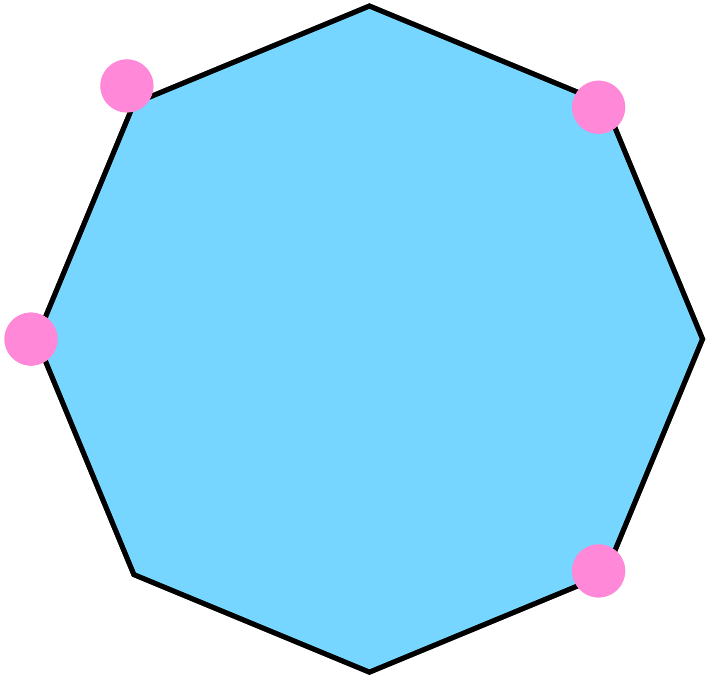
for us: an academic language that
has the qualities of a scripting language,
that is widely used as such, and
that we can change -- if we really must

Granularity of Conversion Steps



- expressions
- procedures
- arbitrary regions of code
- classes
- modules/packages

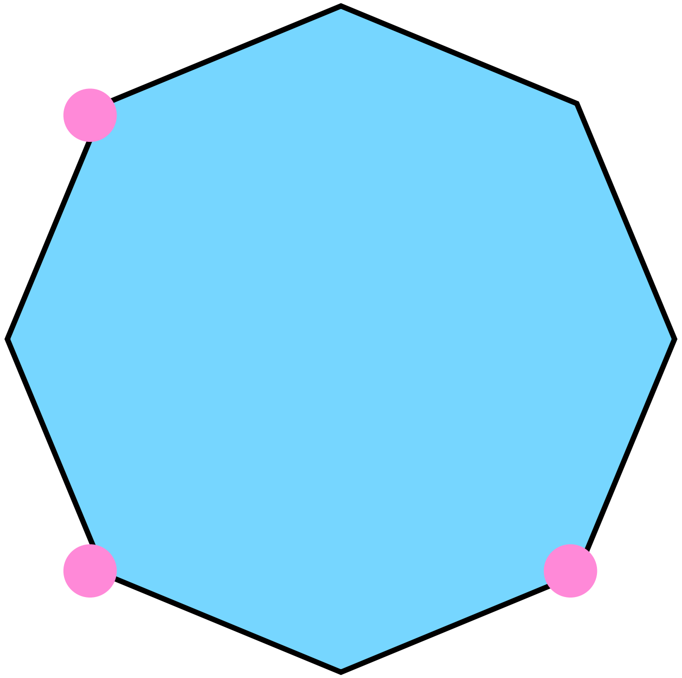
Granularity of Conversion Steps



- expressions
- procedures
- arbitrary regions of code
- classes
- modules/packages

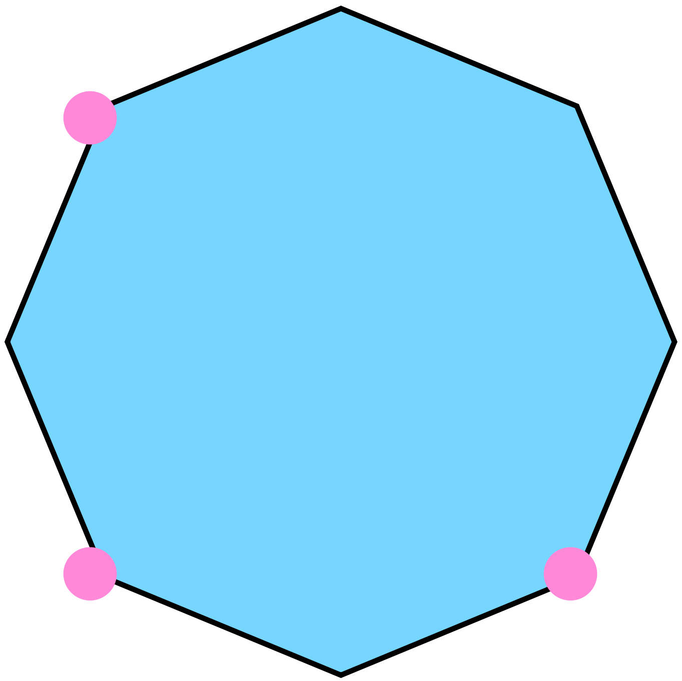
for us: since **soundness** matters to us as well as **performance,** we went with Scheme modules.
In future work, we will also consider units and classes.
Neither of thus demands new instruction sets.

Explicit Types vs Type Inference



- explicit static typing renders design information obvious and checkable
- type inference (HM, SBA, CFA, etc) is brittle, only reports what it sees and doesn't check against specs; error reporting problem isn't solved yet

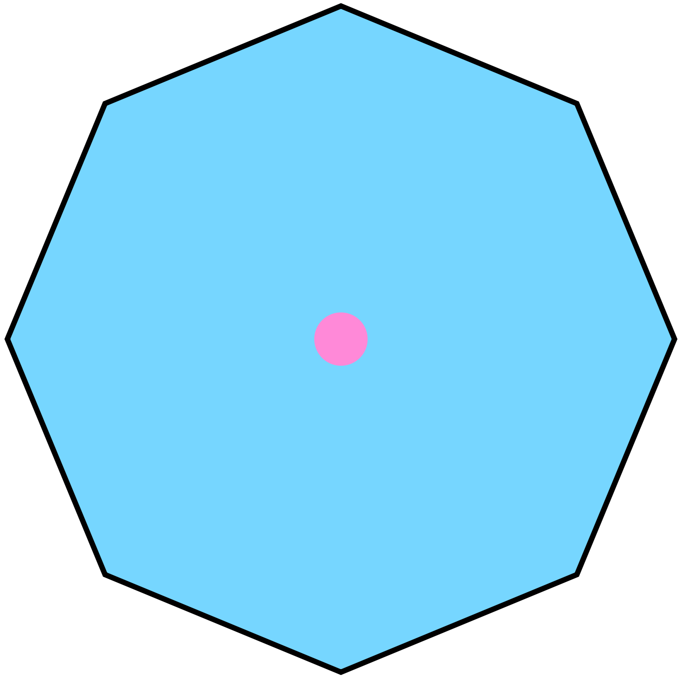
Explicit Types vs Type Inference



- explicit static typing renders design information obvious and checkable
- type inference (HM, SBA, CFA, etc) is brittle, only reports what it sees and doesn't check against specs; error reporting problem isn't solved yet

for us: we go with **local** type inference (for non-rec declarations) to avoid some "finger typing" burden. In the future we will investigate **global** inference as a tool that assists incremental type enrichment.

Quality Level



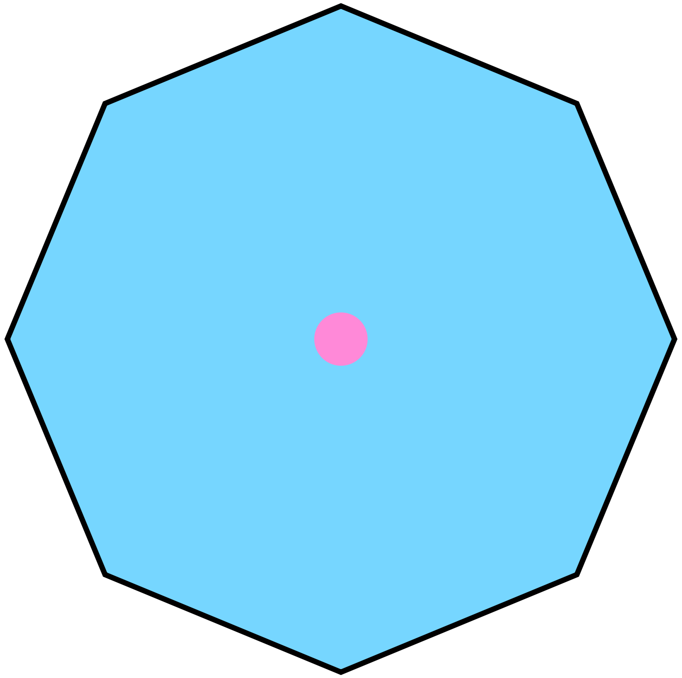
- how much of their code base should "type enrichment" programmers change?

  the size of the code base

  the percentage of conversion

- quality of error feedback

Quality Level



- how much of their code base should "type enrichment" programmers change?

  the size of the code base

  the percentage of conversion

- quality of error feedback

for us: below 3% for changes to code **as opposed to** addition of type information; near immediate understanding of error messages (messages, hyperlinks)

Let’s Go into Details

The End

Thanks to Corky, Mike, Andrew, Cormac, Shriram, Stephanie, Matthew, Robby, Philippe, Sam, Ivan, Stevie,