

Polymorphic Type-Checking in Scheme

Steven L. Jenkins

and

Gary T. Leavens

TR#91-21a

May 1995, Revised May 1996

Keywords: abstract data type, type inference, supertype abstraction.

1995 CR Categories: D.3.3 [*Programming Languages*] Language Constructs -- abstract data types, data types and structures. D.3.4 [*Programming Languages*] Processors -- preprocessors. F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs -- type structure.

Submitted for publication.

(c) Steven L. Jenkins and Gary T. Leavens 1995. Copies may be made for research and scholarly purposes, but not for direct commercial advantage. All rights reserved. Some funding for the project was provided by NSF grant CCR 9593168.

Department of Computer Science

226 Atanasoff Hall

Iowa State University

Ames, Iowa 50011-1040, USA

POLYMORPHIC TYPE-CHECKING IN SCHEME

Steven L. Jenkins
and
Gary T. Leavens
May 5, 1996

Abstract

This paper presents a type-inference system for Scheme that is designed to be used by students in an introductory programming course. The major goal of the work is to present a type system that is simple enough to be used by beginning students, yet is powerful enough to express the ideas of polymorphism, abstract data types (ADTs), and higher-order procedures. The system also performs some rudimentary syntax checking. The system uses subtyping, but only in a primitive fashion. It has a type *datum* which is a supertype of all types, and a type *poof* which is a subtype of all types. It uses intersection types to control the use of *datum* and to generate simple but accurate types.

Acknowledgements

The work of Gary Leavens was supported in part by the National Science Foundation under grant CCR-9593168.

1. INTRODUCTION

This paper presents a type inference system for Scheme that is designed to aid students in an introductory course. While other type-inference systems have been developed for Scheme; for example, STYLE [Lin93] and Soft Scheme [WrC93], the complexity of the types output by these systems is often daunting for beginners. Thus a primary goal for the system is to infer types that are simple enough for students to understand. The textbook used in the course, *Scheme and the Art of Programming* [SpF89] also introduces the concept of abstract data types (ADTs), but implements them directly in Scheme. Hence the use of auxiliary declarations as done in STYLE or Soft Scheme would be unacceptable. Our system uses annotations that are in an auxiliary file, so that the proper use of ADTs can be enforced without changing the Scheme code used in the course text.

While this type systems is simple, it handles a large subset of Scheme, and so its ideas may be more widely applicable.

2. TYPE SYSTEM

This section covers the type system used in this paper. The first subsection discusses the subset of Scheme that the type inference system operates over, and the second subsection presents the type inference rules.

2.1. Domain of the type inference system

This system covers all of R4RS [CR91] except the following:

- declarations of procedures with variable arity
- all but the last expression in a **begin** expression must have a type of *void*
- the second and third arguments to an **if** expression must have the same type
- the first argument to an **if** expression must have type *boolean*
- mutation: **set!**, **vector-set!**, **set-car!**, and **set-cdr!** are not allowed

The first four restrictions are made both to simplify the type-checker and to help simplify students' code. While some of these restrictions are not part of the Scheme language itself, we believe that students produce better code when obeying these restrictions. The last is made to preserve soundness of the type system -- removing the restriction is a problem for future work.

2.2. Type information

This paper represents type information according to the following grammar:

```
<type> ::= number | boolean | string | character | symbol | void | datum | poof
        | <type-variable> | (→ (<type>+ ...) <type>) | (→ (<type>*) <type>)
        | (pair <type> <type>) | (list <type>) | (vector <type>)
        | (and <type>+) | <adt-type>
```

```
<type-variable> ::= A | B | C | D | ... | Z|?1|?2|...
```

```
<adt-type> ::= <symbol>
```


TYPE-INFERENCE RULES

The list of the type inference rules used is given below. Most of the notation follows Cardelli's presentation of type rules for his subset of ML [Car87]. For type environments, the expression $x:\tau$ is the binding of variable x to type τ . If A and B are type environments, then $A \cup B$ is A extended by B . The expression $A \vdash e : \tau$ means that given the type environment A , one can infer that e has type τ . The horizontal bar can be read as "*implies*".

$$[\text{LAMBDA}] \quad \frac{\Gamma \cup \{x_1 : \sigma_1, \dots, x_n : \sigma_n\} \vdash e : \tau}{\Gamma \vdash (\mathbf{lambda}(x_1 \dots x_n) e) : (\rightarrow (\sigma_1 \dots \sigma_n) \tau)}$$

$$[\text{APPL}] \quad \frac{\Gamma \vdash e_0 : (\rightarrow (\sigma_1 \dots \sigma_n) \tau), \Gamma \vdash e_1 : \sigma_1, \dots, \Gamma \vdash e_n : \sigma_n}{\Gamma \vdash (e_0 e_1 \dots e_n) : \tau}$$

$$[\text{APPL}^*] \quad \frac{\Gamma \vdash e_0 : (\rightarrow (\sigma_1 \dots \sigma_k u \dots) \tau), \Gamma \vdash e_1 : \sigma_1, \dots, \Gamma \vdash e_k : \sigma_k, \forall_{k+1} \leq i \leq n (\Gamma \vdash e_i : u)}{\Gamma \vdash (e_0 e_1 \dots e_n) : \tau}$$

$$[\text{BEGIN}] \quad \frac{\Gamma \vdash e : \tau, \Gamma \vdash c_i : \mathit{void}}{\Gamma \vdash (\mathbf{begin} c_1 \dots e) : \tau}$$

$$[\text{DATUM}] \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \mathit{datum}}$$

$$[\text{POOF}] \quad \frac{\Gamma \vdash e : \mathit{poof}}{\Gamma \vdash e : \tau}$$

$$[\text{AND}] \quad \frac{\Gamma \vdash e : (\mathbf{and} \sigma_1 \dots \sigma_n)}{\Gamma \vdash e : \sigma_1, \dots, \Gamma \vdash e : \sigma_n}$$

$$[\text{IDE}] \quad \Gamma \vdash x : \tau, \text{ if } \Gamma(x) = \tau$$

$$[\text{IF1}] \quad \frac{\Gamma \vdash e_0 : \mathit{boolean}, \Gamma \vdash e_1 : \sigma, \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (\mathbf{if} e_0 e_1 e_2) : \sigma}$$

$$[\text{IF2}] \quad \frac{\Gamma \vdash e_0 : \mathit{boolean}, \Gamma \vdash e_1 : \mathit{void}}{\Gamma \vdash (\mathbf{if} e_0 e_1) : \mathit{void}}$$

$$[\text{LET}] \quad \frac{\Gamma' = \Gamma \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\}, \Gamma \vdash e_1 : \tau_1, \dots, \Gamma \vdash e_n : \tau_n, \Gamma' \vdash e : \tau}{\Gamma \vdash (\mathbf{let} ((x_1 e_1) \dots (x_n e_n)) e) : \tau}$$

$$[\text{LETREC}] \quad \frac{\Gamma' = \Gamma \cup \{x_1 : \tau_1, \dots, x_n : \tau_n\}, \Gamma' \vdash e_1 : \tau_1, \dots, \Gamma' \vdash e_n : \tau_n, \Gamma' \vdash e : \tau}{\Gamma \vdash (\mathbf{letrec} ((x_1 e_1) \dots (x_n e_n)) e) : \tau}$$

There is no rule for **define**, which is because **define** is desugared into a **letrec**. Note that the type system does not infer *and*-types or the types of procedures with variable arity, although it can use such types, and does so for Scheme's built-in procedures.

3. TYPE INFERENCE ALGORITHM

The algorithm used to infer types is essentially the **j** algorithm from Milner's original paper on type inference [Mil78], [Car87]. In the following subsections we discuss extensions to the standard algorithm.

3.1 And-types

The system treats *and*-types by doing an ordered search over the possible types of a procedure. The order of the rules is very important. Placing the most restrictive rule first and proceeding to the least restrictive will provide the strongest typing of an expression. For example, the system's built-in type for **cons** is as follows.

$$\begin{aligned} &(\text{and } (\rightarrow (T \text{ (list } T)) \text{ (list } T)) \\ &\quad (\rightarrow (S \text{ (list } U)) \text{ (list datum)}) \\ &\quad (\rightarrow (V W) \text{ (pair } V W))) \end{aligned}$$

The ordered search through the conjuncts of this type ensures that homogeneous list types are inferred when possible. Heterogeneous list types are the second preference, and pair types are only used as a last resort. Hence (**cons 1 '()**) has type *(list number)*, (**cons 3 (cons 'a '())**) has type *(list datum)*, while (**cons 3 4**) has type *(pair number number)*. Suppose the type of **cons** were as follows.

$$\begin{aligned} &(\text{and } (\rightarrow (V W) \text{ (pair } V W)) \\ &\quad (\rightarrow (T \text{ (list } T)) \text{ (list } T)) \\ &\quad (\rightarrow (S \text{ (list } U)) \text{ (list datum)})) \end{aligned}$$

If this were the case, the algorithm would always infer that **cons** creates pairs; because both *V* and *W* are type variables, they would always unify with the types of actual parameters, so the algorithm would never infer that **cons** creates lists. Thus the system would give a weaker and more complex type than necessary. Therefore, care must be taken when declaring the built-in procedures.

3.2 Poof

The type *poof* is the subtype of all types. It is used for the types of procedures that do not return to their caller. In our system, it is only used to denote the return type of **error** and in the type of **call-cc**. Even for an introductory class, however, **error** is important. The example below shows a typical use.

```
(define average
  (lambda (ls)
    (if (null? ls)
        (error "no data!")
        (/ (sum ls) (length ls)))))
```

The type of **average** should be $(\rightarrow (list\ number)\ number)$, thus the type of **error** must be unifiable with *number*, otherwise, the type could not be inferred. However, because the return type of **error** is *poof*, and *poof* is a subtype of *number*, this does work.

3.3 Datum

All other types are subtypes of *datum*. However, it is well-known that having such a type can suppress detection of many type-errors, because every expression could have type *datum*. We control the use of *datum* by never using *datum* unless forced to. With this in mind, only a few procedures should have type *datum*. Its primary uses are in building heterogenous lists (via **cons** or **list**) and producing output (via **writeln** or in **error**). One other use is in the procedure **make-vector**. In our system, **make-vector** has the type

$(and\ (\rightarrow (number\ T)\ (vector\ T))$
 $\quad (\rightarrow (number)\ (vector\ datum)))$

If **make-vector** is passed a length as its only argument, then the vector created has undetermined fill values, hence the use of *datum*.¹

Within the algorithm, any other type or type variable unifies with type *datum*. Thus *number* and *datum* unify to *datum*. However, if *datum* is not one of the types involved, then normal unification takes place.

3.4 Procedures of Variable Arity

While procedures that take a variable number of arguments are handled somewhat, only procedures whose optional arguments are homogenous are handled; for example, the procedure **map-all** defined below can have its type $(\rightarrow ((\rightarrow S\ T)\ ((list\ S)\ \dots))\ (list\ T))$ built into the system, although its type cannot be inferred.

```
(define map-all
  (lambda args
    (cond ((null? args) '())
          ((null? (cdr args)) '())
          (else (apply map-all (list (car args)
                                      ((car args) (cadr args))
                                      (caddr args)))))))
```

One direction this research should take in the future is the addition of a type-inferencing system for these procedures. The work of Dzung and Haynes [DH94] could be used to infer the types of such procedures.

¹This treatment of **make-vector** described here is not entirely satisfactory. The type system could be extended with a better treatment of polymorphic mutable data as in [WrC93]. As it is, a vector of *datum* is practically worthless.

3.4.5. ADTs

A major innovation of this project is handling ADTs. Our goals for the ADT work were to:

- avoid changing the source code, which would confuse students,
- force the use of the ADT's operations and thus prevent direct use of the representation, and
- allow students to define their own ADTs

This is done by adding additional declarations to another file (the **.def** file). To see how these goals have been achieved, we will look at the ADT *ratl*, i.e., rationals. This is the first ADT mentioned in *Scheme and the Art of Programming*[SpF89].

```
(define make-ratl
  (lambda (numr denr)
    (if (zero? denr)
        (error "The denominator cannot be zero.")
        (list numr denr))))
```

```
(define numr
  (lambda (ratl)
    (car ratl)))
```

```
(define denr
  (lambda (ratl)
    (cadr ratl)))
```

The types of these procedures can be inferred, and the types will be as follows:

- *make-ratl*: $(\rightarrow (\text{number number}) (\text{list number}))$
- *numr*: $(\rightarrow ((\text{list } T)) T)$
- *denr*: $(\rightarrow ((\text{list } T)) T)$

However, to hide the fact that *ratls* are represented by lists, a file called **ratl.def**, where **ratl.ss** is the name of the code file, can be used. The system will automatically look for a file **ratl.def** that contains type information both for the representation of the ADT and for the ADT's procedures. For example, the contents of the file **ratl.def** are shown below.

```
(defrep ratl (list number))
(deftype make-ratl (-> (number number) ratl))
(deftype numr (-> (ratl) number))
(deftype denr (-> (ratl) number))
```

Here, **defrep** declares that the implementation of a **ratl** is **(list number)**. Hence, any procedures in this file that are declared by a **deftype** should have all occurrences of **ratl** replaced by **(list number)** in their actual implementation in the **ratl.ss** file. Thus, if the implementation of **numr** in **ratl.ss** is not of a type that unifies with $(\rightarrow ((\text{list number})) \text{number})$, then there is a mismatch between the specification and the implementation. In general, the types inferred for the procedures in the code file must unify with the given types, after the abstract type is replaced by the representation type [MP87].

In the current system, every **.def** file must contain exactly one **defrep** expression. However, this restriction could be relaxed in the future if desired.

4. DISCUSSION

This section discusses how effective the system is in inferring types for code. The section also attempts to place this work in context with other work in developing type systems for Scheme.

4.1. Demonstration of applicability

This section demonstrates that the type inference system is effective over its restricted domain. In most cases the type inference system correctly catches type errors, and correctly inferred the types of syntactically correct procedures.

To demonstrate the applicability of the system, we ran it on the code from the first ten chapters of [SpF89] (SAP) have been checked, as well as sample student code from exercises. Also, code from the first chapter of [AbS85] (SICP) was examined. The procedure for checking this body of code was to type-check each file containing code, and examine the output. Procedures must be typed before they are referred to; thus mutually recursive procedures can cause problems; however, the mutually recursive procedures can be placed into a **letrec** expression, and then the individual procedures type-checked one at a time within the body of the **letrec**. A program, **type-check-file** has been developed that does this dependency handling, but it is limited in its usefulness as the size of files increases. The program reads in a file and attempts to resolve all dependencies; however, if there are more than one version of a program per file (for example, 3 versions of the **factorial** program), then the dependencies do not get resolved properly. However, we renamed procedures to take care of that problem. Procedures of variable arity also cause significant problems: often the type system will *crash* attempting to handle them. Such procedures are removed from the test but noted in the results (as not among those correctly typed). Also, since the programs below were from various sources, some of them relied on implementation-specific, or non-standard Scheme procedures. This problem was easily solved: definitions for the undefined helping procedures were included in the file, but not shown in the results table (so as not to inflate the results).

The table below summarizes the results of using the type inference system over these bodies of Scheme code.

Source of Code	Number of procedures	Number of error messages	Number of procedures incorrectly typed	Percentage of procedures correctly typed
Chapter 2, SAP	11	2	0	82%
Chapter 3, SAP	23	0	0	100%
Chapter 4, SAP	20	0	0	100%
Chapter 5, SAP	27	1	0	96%
Chapter 6, SAP	19	2	1	84%
Chapter 7, SAP	36	3	1	78%*(4 not included)
Chapter 9, SAP	23	1	3	74%*(2 not included)
Student code, SAP	16	3	0	81%
Chapter 1, SICP	101	5	0	95%
Totals	276	17	6	96%

Most problems occurred with recursive types, such as trees, or with **make-vector**.

Chapter 2 of [SpF89] produced 4 errors: three of these were problems with procedures taking advantage of Scheme's using any non-**#f** value to mean true. The only other problem was with a procedure that performed *ad hoc* polymorphism, and, thus, is beyond the scope of our system. The code for that procedure is shown below.

```
(define describe
  (lambda (s)
    (cond
      ((null? s) (quote '()))
      ((number? s) s)
      ((symbol? s) (list 'quote s))
      ((pair? s) (list 'cons (describe (car s)) (describe (cdr s))))
      (else s))))
```

Chapter 3 of [SpF89] produced no errors. Even though much of the procedures deal with the ADT *ratls*, all the procedures built on top of the ADT successfully checked since we included a **.def** file.

Almost all of the problems in chapters 4 and 5 of [SpF89] were caused by procedures that operate over trees. An example is the procedure **remove-all**:

```
(define remove-all
  (lambda (item ls)
    (cond
      ((null? ls) '())
      ((equal? (car ls) item) (remove-all item (cdr ls)))
      ((pair? (car ls))
       (cons (remove-all item (car ls)) (remove-all item (cdr ls))))
      (else (cons (car ls) (remove-all item (cdr ls))))))
```

There are no facilities for recursive types, such as trees, in the type system; thus this program cannot have its type inferred.

In chapter 6 of [SpF89], all of the problems were caused by interactive procedures. For example, the following code produces an error:

```
(define interactive-square
  (lambda ()
    (let ((val (read)))
      (if (eq? val 'done)
          (writeln "Thanks for playing")
          (begin
             (writeln "The square of " val " is " (* val val))
             (interactive-square))))))
```

The error message occurs because **read** returns both a *symbol* (i.e., 'done) and a *number*, but as the program really is unsafe, the error message is correct.

Chapter 7 of [SpF89] had problems with trees, also, and the system's inability to infer types for procedures with variable arity.

Chapter 9 of [SpF89] produced errors due to the use of **make-vector** with only one argument. This produces vectors of unusable types.

From this data it is clear that a large portion of Scheme code used in introductory classes can be successfully type-checked using our system. The major limitation is in the lack of a satisfactory treatment of: the use of recursive types, union types, mutation, and *ad hoc* polymorphism.

4.2. Related work

In this section we attempt to place our type inference system in context with other type systems for Scheme, as well as other functional languages. We will take a brief look at PLEAT [Cur90], STYLE [Lin93], Soft-Scheme [WrC93], and SPS [Wan89], with the emphasis in the section being on STYLE and Soft-Scheme. We will look at three main areas: domain of the type systems, representation of types, and complexity of code.

The work of Curtis [Cur90] provides an example of a type-system for a small, functional language, PLEAT, much in the style of Scheme. However, the types produced by the system are far too complex for beginning students to handle. For example, the procedure **sum**, which takes a list of numbers and returns the sum of the list has type $(\rightarrow (list\ number)\ number)$ in our type system, but in Curtis', it has the type

$$\forall\alpha. (rec\ \beta. [Empty:\ \alpha, NonEmpty:\ \langle hd:\ int, tl:\ \beta \rangle]) \rightarrow int)$$

While to someone familiar with Curtis' presentation of recursion and his type for **cons**, this is understandable, we believe it is beyond beginning students' ability to use. However, his type system is quite rich and complete. It is just not a good fit for our goals.

The Semantic Prototyping System (SPS) of Wand produces simple types that are more usable than Curtis' for students. However, it requires that students enter in types for their procedures, and then attempts to perform a unification with the type defined by the student, and the type of the procedure declared. Using this on top of existing code proves difficult without employing macros to translate procedures with their types into this language. The only other option would be to teach the students the syntax of SPS, in addition to the regular syntax of Scheme. The other major problem with SPS is that it has no facilities for ADTs along the lines of SAP [SpF89].

A type system that is very close to ours is Christian Lindig's STYLE [Lin93]. It offers a type system that operates over all of Scheme, as well as provides a solid type system. Like PLEAT, a major drawback of STYLE is its complex types. For example, a version of **member?**, has type $(A_nv\ (B_nv\ .\ C_nv) \Rightarrow\ bool)$ in Lindig's system, while it has type $(\rightarrow (T\ (list\ T))\ boolean)$ in ours. In his paper, Lindig provides some results from type-checking code from [AbS85], and his system appears quite practical.

Soft Scheme [WrC93] infers types, and, instead of producing error messages on untypable expressions, inserts run-time checks. The system covers all of R4RS Scheme, is very powerful, and, according to the authors, has been shown to perform well. For our purposes it is unusable. The system is quite large, and requires a great amount of overhead to run. Aside from the practical considerations, however, the only problem with the system is that it doesn't handle ADTs in the same manner that [SpF89] does. Instead it uses facilities for defining record structures. These, however, are more suited for more advanced programmers, not students learning about ADTs. Its handling of recursive types and intersection types is based on the work of Fagan [Fa90] and is richer, yet more complicated, than ours. The complexity of the types presented to the user are comparable with those of PLEAT.

5. FUTURE DIRECTIONS

Future work for this system involves a treatment of union types and *ad hoc* polymorphism. This would allow coverage of more of the Scheme code used in introductory courses. One direction is to follow SoftScheme by inserting run-time checks, but it would seem more effective for teaching to have the system propagate the knowledge of type tests (such as **number?**) to the contexts controlled by such type tests. The work of Olin Shivers [Sh88] might be one direction to pursue along these lines.

Other future work involves extending the system to be more practical. One extension is to infer the types of procedures with variable numbers of arguments (following [DH94]). Another is to allow users to infer (or declare in **.def** files) the types of procedures (like **cons**) that require *and-polymorphism*. The procedure **make-vector** should be handled in some fashion, perhaps as in SoftScheme [WrC93]. Some facility for dealing with ADT type generators and recursive types is needed.

Finally, some treatment of mutation is needed. The problem is that mutable storage cannot contain polymorphic values (if the soundness of the system is to be preserved). Unfortunately, in the full Scheme language *all* procedures are stored in mutable cells, meaning either that the full language cannot be handled or that polymorphism has to be abandoned, or that Hindley-Milner polymorphism is inadequate for the task.

6. CONCLUSION

We have designed and implemented a type inference system for Scheme that:

- has types that are sufficiently simple that they can be understood by beginning college students,
- and enforces type-correct use of ADTs.

The main idea for handling ADTs is to put the declaration information added to a separate file. This idea might be useful for other dynamically-typed languages such as Smalltalk.

Key technical aspects of the type system are its use of intersection types and subtyping. Intersection types and the type of all types, *datum*, are not inferred, and this prevents them from getting "out of control" and allowing all code to type-check. Unifying against intersection types in order allows one to have the type system retain information in simple forms where possible. The challenge is to extend this type system to handle more of Scheme without losing this simplicity.

BIBLIOGRAPHY

- [AbS85] Harold Abelson, Gerald Sussman and Julie Sussman, *Structure and Interpretation of Computer Programs*, 13th edition, The MIT Press, Cambridge, Massachusetts.
- [Car87] Luca Cardelli, Basic Polymorphic Typechecking, *Science of Computer Programming*, 8,2 (April 1987).
- [CR91] William Clinger and Jonathan Rees (eds.), *Revised⁴ Report on the Algorithmic Language Scheme*, 1991.
- [Cur90] Pavel Curtis, *Constrained Quantification in Polymorphic Type Analysis*, Xerox PARC Technical Report CSL-90-1, February 1990.
- [DH94] Hsianlin Dzeng and Christopher T. Haynes, Type Reconstruction for Variable-Arity Procedures, 1994, to be published.
- [Fa90] M. Fagan, Soft Typing: An Approach to Type Checking for Dynamically Typed Languages, PhD thesis, Rice University, 1992.
- [Hin69] R. Hindley, The principal type scheme of an object in combinatory logic, *Transactions of the American Mathematical Society*, Volume 146, December 1969, pp. 29-60.
- [Lin93] Christian Lindig, STYLE: A Practical Type Checker for Scheme, Technische Universität Braunschweig, Informatik-Bericht Nr. 93-10, October 1993.
- [MP87] John Mitchell and Gordon Plotkin, Abstract Types have Existential Type, Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, ACM, January 1985, 37-51.
- [Mil78] Robin Milner, A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences*, 17 (1978), 348-375.
- [Sh88] Olin Shivers, Control Flow Analysis in Scheme, Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988, 164-174.
- [SpF89] George Springer and Daniel P. Friedman, *Scheme and the Art of Programming*, MIT Press and McGraw-Hill, 1989.
- [Wa89] Mitchell Wand, Semantic Prototyping System (SPS) Reference Manual, Version 1.4 (Chez Scheme), Northeastern University, 1989.
- [WrC93] Andrew Wright and Robert Cartwright, A Practical Soft Type System for Scheme, Rice University Technical Report, TR93-218, December 6, 1993.