Style
A Practical Type Checker for Scheme

Christian Lindig

Informatik-Bericht Nr. 93-10
Oktober 1993

© Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17
D-38092 Braunschweig
Germany

STYLE —A Practical Type Checker for Scheme

Christian Lindig
Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-38106 Braunschweig

Abstract

This paper describes an new tool for finding errors in R^4RS-compliant Scheme programs. A polymorphic type system in the style of Damas & Milner (1982) with an additional maximum type is used to type Scheme code. Although Scheme is dynamically typed, most parts of programs are statically typeable; type inconsistencies are regarded as hints to possible programming errors. The paper first introduces a type system which is a careful balance between rigorous type safety and pragmatic type softness. An efficient and portable implementation based on order sorted unification in Scheme is then described. We obtained very satisfactory results on realistic programs, including the programs in Abelson, Sussman & Sussman (1985).

1 Introduction

Finding errors in Scheme programs is painful. One major reason is that Scheme is a dynamically typed language: all names in Scheme programs can hold objects of arbitrary, undeclared type which may change during runtime. Unlike most modern languages like Haskell (Hudak, Jones & Wadler 1992) or Standard ML (Harper, Milner & Tofte 1989) all necessary type checks are performed at runtime. This leads to the great expressiveness of Scheme but also makes programs hard to debug.

Dynamically typed programming languages are more expressive than statically typed languages (Fagan 1992) and therefore cannot be typed statically at compile-time in general. But because only small parts of typical Scheme programs rely on dynamic typing, actually most program parts *are* statically typable. Untypable parts may be either caused by code relying on dynamic typing or by programming errors, thus type checking may indicate errors. We present a new tool STYLE[1] that performs static typing for Scheme programs using a polymorphic type system in the style of Damas & Milner (1982). STYLE reports type inconsistencies to the programmer and thus helps to find even subtle errors otherwise only found at runtime. Because Scheme was not defined with a type system and compile-time type checks in mind, the design of a type system is a difficult task: a too restrictive type system will produce too many irrelevant warnings; if the type system is too soft it will not be able to detect serious bugs.

STYLE's type system is intended to be practical and hence is a careful compromise between "too strong" and "too soft" type systems. There are other, more expressive type systems for Scheme (Wright 1992) which are used for code optimization. The inferred types are more complicated - perhaps too complicated to be of practical value for the programmer.

[1] Scheme Type Leakage Explorer

2 Typing Scheme

There are basically two approaches for typing a language like Scheme: data-flow analysis and static typing with a maximum type. Data-flow analysis is an appropriate technique for optimized compilation of dynamically typed languages (Ma & Kessler 1990, Shivers 1991). However, classical data-flow analysis works top-down and intra-procedural and therefore cannot handle higher order functions properly.

Static typing as described by Damas & Milner (1982) assigns one, possibly polymorphic, most general type to each object of a program. Dynamic typing is therefore reflected by polymorphic types. Because dynamically typed languages cannot be typed statically in general, a maximum type is assigned, when no other type will do (Henglein 1992, Gomard 1990). STYLE also implements this approach, because it is well understood and efficient in practice (Kanellakis, Mairson & Mitchel 1991).

2.1 Types for Scheme

STYLE distinguishes the following types for objects in Scheme. It employs a fully parenthesized prefix notation for types like Scheme does for expressions.

num All numerical objects are of type *num*. Type checking cannot further distinguish between *integers*, *rationals*, *reals* and *complex* because these type depend on the actual values and the operations performed on them.

bool Boolean values as returned by predicates have type *bool*. Scheme regards everything *true* different from Symbol #f.

char Type of a single character.

string Type of a string of characters.

symbol Quoted symbols evaluate to literal symbols, typed *symbol*. Scheme lacks syntactical constructs for declaring data structures explicitly. Therefore many informations, like enumerational types, are represented by symbols. However, STYLE does not distinguish different symbols and therefore cannot infer much about complex data structures where symbols are used for tagging.

port Type of a port—ports are used for describing files and other input/output-devices in Scheme.

void The value of some procedures or syntactical constructs are *unspecified* according to the Scheme report (Clinger & Rees 1992). Objects of such value will be of type *void*.

bottom Whenever STYLE cannot infer *one* type for an object it assigns *bottom* to that object. This may be caused either by relying on dynamic typing or by a programming error. So *bottom* is the maximum type in the sense above although its name suggests the opposite. When *bottom* is used inside types of standard procedures it allows to bypass type checking, because *bottom* always is sufficient. This is sometimes necessary, because STYLE cannot provide adequate types for some more elaborated standard procedures like call/cc.

($\rightarrow$ (***seq*** $t_1 \cdots t_{n-1}$) t_n) Function type, describing a procedure taking $n-1$ parameters of type t_1 to t_{n-1} and returning a value of type t_n. Some standard procedures take a variable number of arguments; to reflect this, the last input type, i.e. t_{n-1} may be marked, if it is one of the types above: for example *num** denotes *"any number of arguments of type num"*, *port+* denotes *"optional argument of type port"*. The standard procedure + for example, which takes one or more numerical arguments has type ($\rightarrow$ (*seq num num**) *num*.

(***promise*** t) In Scheme which generally evaluates expression call by value the evaluation of an expression may be delayed. The value returned by delay is called a *promise* which can be evaluated by the standard procedure force. The type (*promise* t) describes such a promise, holding a value of type t.

(***pair*** t_1 t_2) Pairs of values are of type *pair*. Lists, which may be of inhomogeneous type in Scheme are also treated as pairs.

(***vector*** $t_1 \cdots t_n$) Type of a vector of size n.

τ^s Type-variable of sort s. STYLE implements type inference via order sorted unification so all types carry sorts. This is discussed in detail below.

$\forall \tau_1^{s_1}, \cdots, \tau_n^{s_n}.t$ Type-scheme with bounded type variables τ_1 to τ_n of sorts s_1 to s_n. Type instantiation and type abstraction is sort preserving.

Types inferred by STYLE are not allowed to be recursive. So the expressiveness of types is somewhat limited, in particular because there is no type like (*list* τ) denoting a list of elements of type τ. We thought quite a while of allowing recursive types and performed several experiments. Finally, we decided that although recursive types are more expressive and more precise, they are often hard to understand (see section 5.4). As a consequence, it was not possible to implement list types through $(list\ \tau) = nil \mid (pair\ \tau\ (list\ \tau))$.

Lists are allowed to have members of different type in Scheme, unlike lists in ML or Haskell. Because of this STYLE regards lists as pair—the way they are implemented—and there is no special type for lists. Instead the last element of a list, the empty list, is handled specially: STYLE assigns an unused type variable to empty lists. This ensures that the types of two lists are still unifyable, when one list is a prefix of the other. Thus STYLE only infers the types from the head of each list, approximating the rest with a type variable. This is surely not very exact, but sufficient for detecting most errors.

2.2 Types carry Sorts

STYLE uses order sorted types to put some constraints on types which otherwise cannot be expressed. A *sort hierarchie* is a partial ordered set of sorts $(S, \leq)$ forming a meet-semilattice (figure 1). Thus for each two sorts s_1, s_2 there exists exactly one sort $s_1 \wedge s_2$ (*"s_1 meets s_2"*) which is the greatest lower bound of s_1 and s_2.

In contrast to sort-free terms and variables, order-sorted variables and terms always have a sort. For all non-variable terms the outermost constructor denotes their sort implicitly: a *pair* is of sort *pair*, a type *num* is of sort *num* and so forth. These sorts are located below the dashed line in figure 1. Type variables and bounded type variables of type schemes

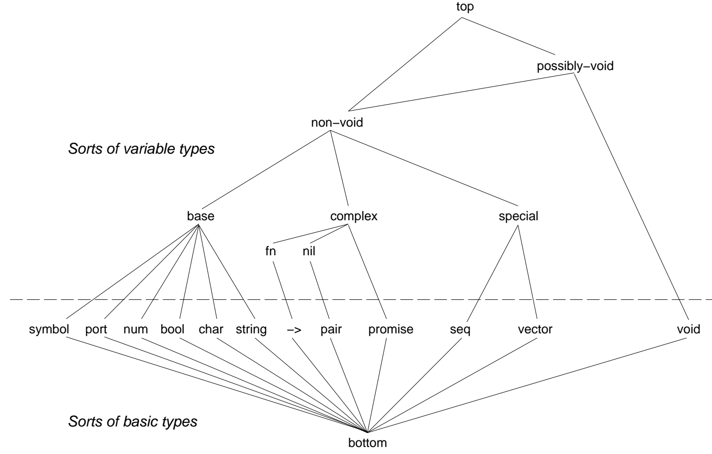

Figure 1: Sort hierarchy of STYLE types

carry sorts from above that line. As a consequence there is more than one kind of type variable in STYLE: $\tau^{complex}$ and τ^{base} denote both variables, but of different kind, thus sort.

Unlike sort-free variables sorted variables do not represent an arbitrary type: a sorted variable τ^s represents only types t of sort s_t where $s_t \leq s$ holds. That is, a sorted variable can only hold terms which sort is localized "below" the sort of that variable: A variable of sort *non-void* cannot hold type *void*, a variable of sort *complex* can hold a type *pair*, but cannot hold type *bool*. A variable of sort *top* can hold any other type and thus is equivalent to a sort-free type variable. Unification of sorted terms must take variable sorts into account (Walther 1985, Meseguer & Goguen 1990). Order-sorted unification has succesfully been used in other type systems (Snelting 1991, Nipkow & Snelting 1991); its main advantage is that many context conditions can be expressed as sortal constraints.

2.3 A Type System for Scheme

The type system gives one rule for each syntactical construct determining its type, while types of standard procedures are part of the initial type assumption Γ_0 (figure 2). As an example of the latter, consider the rule for $reverse$: since lists may be heterogenuous, we can only enforce that $reverse$ works on pairs whose elements must have a non-void type; there are no constraints on the type of the pair components itself .

$$\begin{array}{ll}
abs: & (\rightarrow\ (seq\ num)\ num) \\
display: & \forall\tau^{non-void}.(\rightarrow\ (seq\ \tau^{non-void}\ port+)\ void) \\
not: & \forall\tau^{non-void}.(\rightarrow\ (seq\ \tau^{non-void})\ bool) \\
reverse: & \forall\tau_1^{non-void},\tau_2^{non-void},\tau_3^{non-void},\tau_4^{non-void}.(\rightarrow\ (seq\ (pair\ \tau_1^{non-void}\ \tau_2^{non-void})) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad (pair\ \tau_3^{non-void}\ \tau_4^{non-void})) \\
force: & \forall\tau^{non-void}.(\rightarrow\ (seq\ (promise\ \tau^{non-void}))\ \tau^{non-void})
\end{array}$$

Figure 2: Types for standard procedures

$$\textsc{Const-Num} \quad \frac{c \text{ is a number}}{\Gamma \vdash c : num} \qquad \textsc{Quote-Sym} \quad \frac{c \text{ is a symbol}}{\Gamma \vdash (\mathsf{quote}\ c) : symbol}$$

$$\textsc{Const-Null} \quad \frac{c = ()}{\Gamma \vdash c : \beta^{nil}} \qquad \textsc{Quote-Null} \quad \frac{c = ()}{\Gamma \vdash (\mathsf{quote}\ c) : \beta^{nil}}$$

$$\textsc{Var} \quad \frac{x : \tau \in \Gamma, \tau \preceq \tau'}{\Gamma \vdash x : \tau'} \qquad \textsc{Begin} \quad \frac{\Gamma \vdash e_1 : \tau, \cdots, \Gamma \vdash e_n : \tau}{\Gamma \vdash (\mathsf{begin}\ e_1 \ \cdots \ e_n) : \tau_n}$$

$$\textsc{And} \quad \frac{\Gamma \vdash e_1 : \tau_1, \cdots, \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (\mathsf{and}\ e_1 \ \cdots \ e_n) : \tau_n} \qquad \textsc{Or} \quad \frac{\Gamma \vdash e_1 : \tau, \cdots, \Gamma \vdash e_n : \tau}{\Gamma \vdash (\mathsf{or}\ e_1 \ \cdots \ e_n) : \tau}$$

$$\textsc{Delay} \quad \frac{\Gamma \vdash c : \tau}{\Gamma \vdash (\mathsf{delay}\ c) : (\text{promise } \tau)} \qquad \textsc{Set} \quad \frac{\Gamma \vdash var : \tau, \Gamma \vdash e : \tau}{\Gamma \vdash (\mathsf{set!}\ var\ e) : void}$$

$$\textsc{Cond-Clause} \quad \frac{\Gamma \vdash e_0 : \tau_0, \cdots, \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_0 \ \cdots \ e_n) : \tau_n} \quad \frac{\Gamma \vdash e_1 : \tau_1, \cdots, \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (\mathsf{else}\ e_1 \ \cdots \ e_n) : \tau_n}$$

$$\textsc{Cond} \quad \frac{\Gamma \vdash c_1 : \tau, \cdots, \Gamma \vdash c_n : \tau}{\Gamma \vdash (\mathsf{cond}\ c_1 \ \cdots \ c_n) : \tau} \quad c_i \text{ typed using } \textsc{Cond-Clause}$$

$$\textsc{If} \quad \frac{\Gamma \vdash e_0 : \tau_0, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{if}\ e_0\ e_1\ e_2) : \tau} \quad \frac{\Gamma \vdash e_0 : \tau_0, \Gamma \vdash e_1 : \tau}{\Gamma \vdash (\mathsf{if}\ e_0\ e_1) : \tau}$$

$$\textsc{Lambda} \quad \frac{\Gamma \cup \{x_1 : \tau_1, \cdots, x_n : \tau_n\} \vdash \langle\text{body}\rangle : \tau}{\Gamma \vdash (\mathsf{lambda}\ (x_1 \ \cdots \ x_n) \langle\text{body}\rangle) : (\rightarrow \ (seq\ \tau_1 \ \cdots \ \tau_n)\ \tau)}$$

$$\textsc{Let*} \quad \frac{\begin{array}{c}\Gamma_1 = \Gamma, \Gamma_i = \Gamma_{i-1} \cup \{x_{i-1} : ta(\tau_{i-1}, \Gamma_{i-1})\}, \\ \Gamma_1 \vdash e_1 : \tau_1, \cdots, \Gamma_n \vdash e_n : \tau_n, \Gamma_{n+1} \vdash \langle\text{body}\rangle : \tau\end{array}}{\Gamma \vdash (\mathsf{let*}\ ((x_1\ e_1) \cdots (x_n\ e_n))\ \langle\text{body}\rangle) : \tau}$$

$$\textsc{Letrec} \quad \frac{\begin{array}{c}\Gamma' = \Gamma \cup \{x_1 : \tau_1, \cdots, x_n : \tau_n\}, \Gamma' \vdash e_1 : \tau_1, \cdots, \Gamma' \vdash e_n : \tau_n, \\ \Gamma \cup \{x_1 : ta(\tau_1, \Gamma), \cdots, x_n : ta(\tau_n, \Gamma)\} \vdash \langle\text{body}\rangle : \tau\end{array}}{\Gamma \vdash (\mathsf{letrec}\ ((x_1\ e_1) \cdots (x_n\ e_n))\ \langle\text{body}\rangle) : \tau}$$

$$\textsc{Appl} \quad \frac{\Gamma \vdash e_0 : (\rightarrow \ (seq\ \tau_1 \ \cdots \ \tau_n)\ \tau_0), \Gamma \vdash e_1 : \tau_1, \cdots, \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_0\ e_1 \ \cdots \ e_n) : \tau_0}$$

$$\textsc{Quote-Pair} \quad \frac{c = (e_1\ .\ e_2), \Gamma \vdash (\mathsf{quote}\ e_1) : \tau_1, \Gamma \vdash (\mathsf{quote}\ e_2) : \tau_2}{\Gamma \vdash (\mathsf{quote}\ c) : (\text{pair } \tau_1\ \tau_2)}$$

$$\textsc{Def-Simple} \quad \frac{\Gamma \vdash e : \tau, \Gamma' = \Gamma \cup \{x : ta(\tau, \Gamma)\}}{\Gamma \vdash (\mathsf{define}\ x\ e) : void}$$

Figure 3: Rules from the type system for Scheme

A Milner-style type system assigns to each object one, possibly polymorphic type. This does not reflect dynamic typing exactly and therefore STYLE's type system only approximates the real behavior of Scheme. We present now a selection from the typing rules for Scheme as implemented in STYLE (figure 3).

Type instantiation is denoted by $\tau \preceq \tau'$: τ' is an instance of the type scheme τ. Type abstraction, which is defined relative to the type assumption Γ is denoted by $ta(\tau, \Gamma)$, where $FV(x)$ denotes the set of free type variables of x:

$$ta(\tau, \Gamma) = \forall \nu_1, \ldots, \nu_m.\tau \quad \nu_i \in FV(\tau) \setminus FV(\Gamma)$$

Both type instantiation and abstraction are sort preserving. This permits to detect the application of undefined values (of type *void*) to standard procedures: bounded type variables inside types of standard procedures usually have sort *non-void* (figure 2). These variables can not hold type *void* and thus STYLE detects every application of undefined values to the corresponding parameters for example in `(define x (set! y 10))`.

While some rules are fairly obvious (CONST-NUM, APPL, QUOTE-SYM, DELAY, LAMBDA) others need some explanation. For example an **and**-expression returns the value of its last argument or #**f**. Contrary to statically typed languages Scheme regards **and**- and **or**-expressions as *conditionals*. The type system reflects this, by typing the whole expression after the type of the last argument to **and**. An **or**-expression on the other hand returns #**f** or the value of one of its arguments. So STYLE enforces all arguments of **or** to be of the same type. Enforcing all arguments to **and** and **or** to be of type *bool* would have been closer to ML-like languages but would cause irrelevant warnings.

Whenever Scheme checks an expression for a boolean value it regards everything *true* different from #**f**. So the type system does not expect a type of such an expression to be *bool* but puts no constraint on that type (COND-CLAUSE, IF). Typical Scheme programs rely on that feature very often and therefore enforcing that expression to be of type *bool* would not model the real behavior of Scheme.

Expressions returning different values depending on some conditions are forced to return always objects of the same type (IF, COND, CASE) and the expression itself is assigned that type, too. Expressions returning values of different types depending on some condition are not uncommon in Scheme. However, we believe that each expression should have *one* type, reflecting an abstraction of its value. We regard inconsistent **if**- or **cond**-expression as one major hint to possible errors.

The rules for **let** and **let*** look very complicated, but they are not, indeed: **let*** enlarges the scope successively by each defined name and therefore the description of Γ has to reflect this. A **letrec**-expression in Scheme allows multiple names to be defined recursively. The defined names are allowed to be polymorphic inside the body of **letrec**, but not inside the definitions of the names. STYLE treats definitions by **define**, **let**, **let*** and **letrec** as polymorphic.

Because Scheme supports top-level and local definitions a program is not just one big expression but consists of multiple, independent expressions. Therefore STYLE assigns type *void* to definitions (SIMPLE-DEFINE), and enters the defined name to the type assumption Γ, resulting in Γ'. The enlarged type assumption Γ' now must be used, to type all expression, which rely on the defined name.

The rules CONST-NULL and QUOTE-NULL show that empty lists are treated specially in STYLE: empty lists are treated as variables of sort *nil*. That is, they can hold objects of sort *nil* or *pair* (and *bottom* of course). Hence lists can only be enlarged by pairs, rather than by objects of arbitrary type.

3 Implementation

Style is intended as a practical tool and therefore some effort went into its portable and efficient implementation. This section gives a brief overview how Style is implemented. First of all Style is implemented in Scheme to ensure portability. Furthermore the implementation requires only a Scheme implementation conforming to Clinger & Rees (1992). It is based on SCM (Jaffer 1993*b*) and uses few procedures from the Scheme library SLib (Jaffer 1993*a*) for pretty printing. The current implementation supports the checking of nearly all R4RS-compliant Scheme code except macros.

3.1 Syntax Check

Before Scheme code can be type checked its syntax must be checked for correctness. Because some Scheme implementations perform only a weak syntax check (Jaffer 1993*b*, Carrette 1992) this step helps to find a lot of errors in practice. The implemented parser utilizes the standard procedure `read` for reading in Scheme source code. Unfortunately this is insufficient to parse ill-parenthesed code; alternatively a hand written scanner as implemented by Sclint (Kellomäki 1992) could have been used.

The syntax checks in Style detects unbound or multiple defined names and makes all bindings visible to the programmer by renaming identifiers to be unique. Because `quasiquote`-expressions have a complicated structure they are rewritten to semantically equivalent expressions which are easier to type check later on. This technique is also used by Tammet (1993) and Wright (1993).

3.2 Dependency Analysis

A Scheme program is not just one big expressions but consists of many independent definitions, so type checking is not purely syntax directed. Typing a Scheme program requires to sort independent definitions (local or top-level) and order them accordingly to their dependencies (Diller 1988). Figure 4 shows a simple example illustrating this together with the corresponding dependency graph.

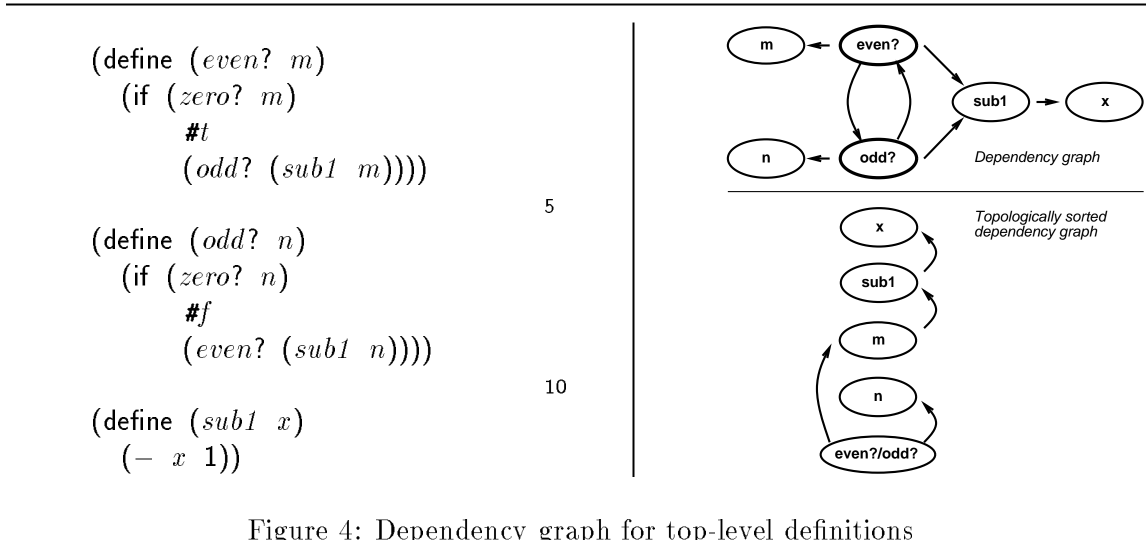

Figure 4: Dependency graph for top-level definitions

Strongly connected components are merged and the resulting graph is topologically sorted. The order of the nodes determines the order in which the definitions are typed. Each typed definition enlarges the type assumption for subsequent typings. Mutual recursive definitions are handled the same way as the defining expressions inside a **let**-expression.

3.3 Unification

Typing Scheme in STYLE bases on the algorithm presented by Damas & Milner (1982) and hence on unification and generic instantiation. Unification must take the sort carried by each type into account and thus is order sorted. We do not want to present a complete algorithm for order sorted unification (Meseguer & Goguen 1990), but give a sketch of the main idea:

- Two types of sorts s_1 and s_2 are unifyable if $s_1 \wedge s_2 \neq \mathit{bottom}$ holds and both types are syntactically unifyable.
- Two variables τ_1^x, τ_2^y unify: $\tau_1^x, \tau_2^y \mapsto \tau_3^{x \wedge y}$.

In addition, unification in STYLE implements another rule: whenever one of the two unificands already is *bottom*, unification will succeed resulting in an empty substitution. This ensures that type *bottom* will not propagate during typing, but stays local to the corresponding term which caused it. For efficiency STYLE does not implement the classical algorithm of Robinson (1965) but an order sorted variant of the algorithm presented by Corbin & Bidoit (1983).

3.4 Efficient Type abstraction

Naive type abstraction (denoted by $ta(\tau, \Gamma)$ in section 2.3) is not efficient: determining the variables occurring free in τ but not free in Γ requires to search through a possibly large type assumption Γ. An efficient algorithm for type abstraction which is also implemented in STYLE can be found in Rémy (1992): each type variable is assigned a depth which denotes the static depth where it was created. Unification of two variables results in a new variable carrying the minimum depth of the two given variables. The typing algorithm also uses a counter which denotes the actual static depth. Comparing the depth of a variable with the actual depth tells in constant time whether a variable can be made generic.

4 Practical Results

Currently there exist two variants of STYLE: a batch version to be run in an Unix environment and an interactive version, which is simply a large file of Scheme code to be loaded into a Scheme interpreter. Figure 5 shows the output of the latter while checking a file of Scheme code. Generic type variables are denoted by uppercase letters; the attached `_nv` denotes the sort *non-void*.

To investigate the practical relevance of STYLE we applied it to the programs in *Structure and Interpretation of Computer Programs* by (Abelson et al. 1985). The code from each chapter was checked as one program. The results are shown in table 1. Both the number of warnings and the number of types including *bottom* sub-types are moderate in relation to the number of defined objects. Even in chapter three, which contains the functions hardest

```
> (check-file "demos/example.scm")
Log file demos/example.scm.delta opened
demos/example.scm loaded
--  writing source ... --
-- finished  --
Syntax of demos/example.scm successfully checked
fak_2:  (num => num)
my-member?_3:  (A_nv (B_nv . C_nv) => bool)
warning all clauses of cond should have equal type in
(cond ((symbol? x_8) "it's a symbol")
      ((number? x_8) "it's a number")
      (else #f))

type-of_4:  (A_nv => bottom)
type error 3 not a function in
(3)
There are no undefined symbols
type of demos/example.scm: num
All output written to demos/example.scm.delta
Logfile demos/example.scm.delta closed
;Evaluation took 483 mSec (150 in gc) 34193 cons work
#<unspecified>
>
```

Figure 5: STYLE checking a program

to type-check, 78% of all defined objects obtained usable types. Objects which have *bottom* types are usually the consequence of conditionals where the branches have different types (typically there are some "ordinary" branches and one "exception" branch) - that is, *bottom* types are indeed the result of dynamic typing.

As a second example, we checked solutions to exercises of a first year Scheme course held at our university. The solutions consist of six files with together about 1600 lines of code. Although the checked files were programmed by an experienced Scheme programmer, STYLE revealed 12 warnings. Most of the warnings were caused by syntactical incorrectness and only a minor number by violating typing rules. Hand inspection of these warnings revealed that some of them were due to use of dynamic typing, but that - contrary to the programs in (Abelson et al. 1985) - usually a type error is indeed a hint to a programming error. Hence STYLE types are sufficiently precise to locate errors, but still understandable by the programmer.

Checking the six files gives an idea of the speed of STYLE (Table 2). For comparison the same code was checked by Sclint (Kellomäki 1992), a tool for checking Scheme code discussed below. Both ran on a SPARCstation ELC using the Scheme interpreter SCM (Jaffer 1993*b*).

The times from the table show that using STYLE is indeed practicable because its analysis only take a moderate amount of time.

Chapter	Sections[1] (Files)	Defs.[2]	Mult. Defs.[3]	Syntax Errors[4]	Types incl. *bottom*[5]		Warnings
1	14	101	12	0	0	0 %	1
2	16	187	24	1	7	10.4 %	17
3	13	197	27	0	43	21.8 %	36
4	18	163	6	9	12	7.4 %	10
5	16	115	5	5	10	8.7 %	35

[1]The code from each section was put together into one file. [2] Global and local definitions. STYLE determined one type for each definition. [3] Some objects are multiple defined for teaching purposes. [4] Errors with respect to the syntax accepted by STYLE. [5] Each type of a definition which containing at least one sub-type *bottom* counts as one.

Table 1: Results from Checking *Structure and Interpretation of Computer Programs*

Exercise	Lines	STYLE	Sclint	Exercise	Lines	STYLE	Sclint
1	109	1.5 s	10.7 s	4	215	4.7 s	13.1 s
2	211	5.3 s	15.3 s	5	470	18.2 s	33.0 s
3	352	8.5 s	17.8 s	6	231	5.6 s	13.2 s

Table 2: Some Execution Times (in seconds) of STYLE and Sclint

5 Related Work

5.1 Sclint

Sclint by Kellomäki (1992) is an acronym for *Scheme Lint* and reminds of the well known *C program verifier Lint*. It aims mainly at the same targets as STYLE although is does no type checking. Sclint checks the syntax of most syntactical constructs, but unlike STYLE not the more complicated ones like `do` and `quasiquote`. Furthermore it checks the number of applied arguments to procedures and takes care of a correct indentation. Like STYLE it is implemented in Scheme but uses a manually implemented scanner for reading source code. This permits even ill-parenthesized to be checked but slows down checking noticeably, even without type checking. Because Sclint does no type checking it cannot detect a lot of errors in principle.

5.2 Semantic Prototyping System—SPS

The Semantic Prototyping System by Wand (1989) is not intended as a tool for finding errors, but does type checking for a dialect of Scheme. SPS supports rapid prototyping of programming languages and lets the denotational semantic of a prototype be defined using a dialect of Scheme. This subset is intended to be referential transparent and thus does not rely on dynamic typing. SPS type checks using a a polymorphic type system (without a maximum type) and gives detailed error messages. However, SPS cannot deduce which type variables from a type are generic—the user has to declare them explicitly. So SPS addresses only a subset of Scheme which requires additional type declarations.

5.3 Dynamic Typing

The approach of *Dynamic Typing* (Henglein 1992) uses in principal the same techniques as Style, but does not aim at a general tool for finding errors: Instead, inferred types are used to optimize the compilation of Scheme code. Wherever the type of an object is known, run time type checks can be omitted. Henglein (1992) uses a polymorphic type system with a maximum type *dynamic* to type Scheme code. His paper mainly shows the principles of this approach by typing a dynamically typed lambda-calculus with boolean values; the relation to "real" Scheme is not always clear. Wright (1992) investigated a prototype of *Dynamic Typing* by running it on its own code and reports that about 50% of all globally defined names had type *dynamic* or $dynamic \rightarrow dynamic$. Compared to Style, this is obviously not very informative.

5.4 Soft Types for Scheme

The approach of *Soft Types for Scheme* by Wright (1992) is the most sophisticated approach for typing Scheme known to us. Types are both used for finding errors and for optimized compiling of Scheme code. Wherever necessary, explicit type tagging code is added to the original source and thus run time type checking can be omitted. *Soft Typing for Scheme* distinguishes code which needs run time type checking (and thus may be correct) and code which is definitely ill-typed (and thus wrong). It uses a Milner-style polymorphic type system with recursive and union types based on the work of Fagan (1992). Whenever an object is used with two disjoint types it will be assigned the union of these types. This leads to a very expressive type system which produces very exact but complicated types which are hard to understand by the programmer.

In figure 6 union types are denoted by $(+\ t_1\ t_2 \ldots)$, recursive types by $(MU\ \ :x\ t)$ with $:x$ appearing inside t. The figure shows a type inferred by *Soft Typing for Scheme* for a procedure which evaluates simple boolean expressions; it is from the third exercise mentioned above. For comparison the much simpler type inferred by Style is also shown; `327_nv` denotes a type variable of sort *non-void*.

Soft-Typing for Scheme:
```
(eval-bool:
   (let ([:1
     (MU :1 (+ false true sym
     (MU :2 (cons
     (+ false true sym :2) :1))))])
   (:1 -> :1)))
```

Style:
```
eval-bool:  (327_nv => symbol)
```

Figure 6: Types inferred by *Soft-Typing for Scheme* and Style

The implementation of *Soft Types for Scheme* is efficient and uses structure sharing and the efficient type abstraction algorithm by Rémy (1992) like Style, too. This approach was developed independent of Style and goes beyond its aims. We nevertheless believe that the inferred types are too complicated to be useful for detecting errors.

5.5 Conclusion

Style is a practical tool for finding errors in Scheme programs based on a polymorphic type system. The type system is a compromise between a soft and a strong type check to detect serious bugs on one hand but not do produce too much irrelevant warnings on the other. Although dynamically typed languages cannot be typed statically in general the chosen approach is practical—violations of typing rules are regarded as hints to *possible* errors. The implementation of the type system is based on order sorted unification. This makes it possible to detect the application of undefined values to polymorphic procedures and gives type variables a finer semantic. The implementation of Style in Scheme is portable and efficient—checks performed by Sclint (Kellomäki 1992), a tool for syntax checking without type checking take noticeably longer. Tests have shown the practical relevance of the chosen approach.

Style is available by anonymous ftp through [final version will include ftp address].

Acknowledgements. Gregor Snelting originally proposed the development of Style as a support tool for teaching and provided helpful comments on an earlier version of this paper.

References

Abelson, H., Sussman, G. & Sussman, J. (1985), *Structure and Interpretation of Computer Programs*, 13th edition, The MIT Press, Cambridge, Massachusetts, USA.

Carrette, G. (1992): *SIOD—Scheme In One Defun*, Version 2.8, Paradigm Associates Incorporated, Cambridge, Massachusetts, USA.

Clinger, W. & Rees, J. (1992): Revised[4] Report on the Algorithmic Language Scheme, *LISP Pointers* **IV**(3), 1–55.

Corbin, J. & Bidoit, M. (1983): A Rehabilitation of Robinson's Unification Algorithm, *in* R. E. A. Mason, ed., Information Processing 83, Elsevier Science Publishers (North-Holland), pp. 909–914.

Damas, L. & Milner, R. (1982): Principal type schemes for functional programs, *in* Proc. of the 9th POPL, pp. 207–212.

Diller, A. (1988): *Compiling Functional Languages*, John Wiley & Sons Ltd, Chichester, New York Brisbane, Toronto, Singapore, Dependency Analysis, pp. 157–161.

Fagan, M. (1992): Soft Typing: An Approach to Type Checking for Dynamically Types Languages, PhD thesis, Rice University, Houston, Texas, USA.

Gomard, C. K. (1990): Partial Type Inference for Untyped Functional Programs, *in* Proc. of the 1990 LFP, ACM, pp. 282–287.

Harper, R., Milner, R. & Tofte, M. (1989): The Definition of Standard ML, Version 3, Technical Report ECS-LFCS-89-81, Dept. of Computer Science, Univ. of Edinburgh, Edinburgh EH9 3JZ Scotland.

Henglein, F. (1992): Dynamic Typing, *in* B. Krieg-Brückner, ed., ESOP '92—4th European Symposium on Programming, Springer-Verlag, pp. 233–253.

Hudak, P., Jones, S. P. & Wadler, P. (1992): Report on the Programming Language Haskell, Report, University of Glasgow, Dep. of Computing Science, University of Glasgow.

Jaffer, A. (1993*a*), *Scheme Library Slib*, 84 Pleasant Street, Wakefield MA 01880, USA.

Jaffer, A. (1993*b*), *Scheme Release SCM*, 84 Pleasant Street, Wakefield MA 01880, USA.

Kanellakis, P. C., Mairson, H. G. & Mitchel, J. C. (1991): Unification and ML-Type Reconstruction, *in* J.-L. Lassez & G. Plotkin, eds, Computational Logic, MIT-Press, Cambridge, Massachusetts, pp. 444–478.

Kellomäki, P. (1992): Sclint—a Lint-like Program for Scheme, Technical report, Software Systems Lab, Tampere University, Finland.

Lindig, C. (1993): Style - ein Typ-Checker für Scheme, Master's thesis, Technische Universität Braunschweig, Institut für Programmiersprachen und Informationssysteme, Arbeitsgruppe Softwaretechnologie. (in German).

Ma, K. & Kessler, R. R. (1990): TICL—A Type Inference System for Common Lisp, *Software—Practice and Experience* **20**(6), 593–623.

Meseguer, J. & Goguen, J. A. (1990): Order-Sorted Unification, *in* C. Kirchner, ed., Unification, Academic Press, London, pp. 457–487.

Nipkow, T. & Snelting, G. (1991): Type Classes and Overloading Resoloutin via Order-Sorted Unification, *in* Proc. of the 5th ACM Conference of Functional Programming Languages and Computer Architecture, Springer Verlag, pp. 1–14.

Rémy, D. (1992): Extension of ML Type System with a Sorted Equational Theory on Types, Rapports de recherche, INRIA.

Robinson, J. A. (1965): A machine-oriented logic based on the resolution principle, *Journal of the ACM* **12**(1), 23–41.

Shivers, O. (1991): Data-Flow Analysis and Type Recovery in Scheme, *in* P. Lee, ed., Topics in Advanced Language Implementation, MIT Press, Cambridge, Massachusetts, pp. 47–88.

Snelting, G. (1991): The calculus of context relations, *Acta Informatica* **28**, 411–445.

Tammet, T. (1993): *Hobbit*, Version 2, Dep. of Computer Science, Chalmers University of Technology, University of Göteborg, S-41296 Göteborg, Sweden.

Walther, C. (1985): Unification in many-sorted theories, *in* Proceedings of the 6th ECAI, pp. 593–602.

Wand, M. (1989): *Semantic Prototyping System (SPS) Reference Manual*, Version 1.4 (Chez Scheme), Northeastern University, USA.

Wright, A. K. (1992): Practical Soft Typing: Types for Scheme, Thesis Proposal; Dep. of Computer Science, Rice University, Houston, Texas, USA.

Wright, A. K. (1993): Practical Soft Typing: Types for Scheme, Personal communications.