A Practical Soft Type System for Scheme

Andrew K. Wright* Robert Cartwright†

Department of Computer Science
Rice University
Houston, TX 77251-1892
{wright,cartwright}@cs.rice.edu

Abstract

Soft typing is a generalization of static type checking that accommodates both dynamic typing and static typing in one framework. A soft type checker infers types for identifiers and inserts explicit run-time checks to transform untypable programs to typable form. *Soft Scheme* is a practical soft type system for R4RS Scheme. The type checker uses a representation for types that is expressive, easy to interpret, and supports efficient type inference. Soft Scheme supports all of R4RS Scheme, including uncurried procedures of fixed and variable arity, assignment, and continuations.

1 Introduction

Dynamically typed languages like Scheme [6] permit program operations to be defined over any computable subset of the data domain. To ensure safe execution,[1] primitive operations confirm that their arguments belong to appropriate subsets called *types*. The types enforced by primitive operations induce types for defined operations. Scheme programmers typically have strong intuitive ideas about the types of program operations, but dynamically typed languages offer no tools to discover, verify, or express such types.

Static type systems like the Hindley-Milner type discipline [11, 16] provide a framework to discover and express types. Static type checking detects certain errors prior to execution and enables compilers to omit many run-time type checks. Unfortunately, static type systems inhibit the freedom of expression enjoyed with dynamic typing. To ensure safety, programs that do not meet the stringent requirements of the type checker are ineligible for execution. In rejecting untypable programs, the type checker also rejects meaningful programs that it cannot prove are safe. Equivalent typable programs are often longer and more complicated.

Soft typing [5, 7] is a generalization of static type checking that accommodates both dynamic typing and static typing in one framework. Like a static type checker, a soft type checker infers syntactic types for identifiers and expressions. But rather than reject programs containing untypable fragments, a soft type checker inserts explicit run-time checks to transform untypable programs to typable form. From the perspective of dynamic typing, soft typing recovers type information and globally optimizes run-time checking at primitive operations. From the perspective of static typing, soft typing allows programmers to develop prototypes using a purely semantic understanding of types as meaningful sets of values. Prototypes can then be transformed to more robust, maintainable, and efficient programs by rewriting them to accommodate better syntactic type assignment.

We have developed a practical soft type system for R4RS Scheme [6], a modern dialect of Lisp. *Soft Scheme* is based on an extension of the Hindley-Milner polymorphic type discipline that incorporates recursive types and a limited form of union type. *Soft Scheme* requires no programmer supplied type annotations and presents types in a natural type language that is easy for programmers to interpret. Type analysis is sufficiently accurate to provide useful diagnostic aid to programmers: our system has detected several elusive errors in its own source code. The type checker typically eliminates 90% of the run-time checks that are necessary for safe execution without soft typing. We have observed soft typed programs to run up to 3.3 times faster than ordinary dynamically typed programs.

The type system underlying Soft Scheme is a refinement and extension of a soft type system designed by Cartwright and Fagan for an idealized functional language [5, 7]. Their system extends Hindley-Milner typing with limited union types, recursive types, and a modicum of subtyping as subset on union types. Soft Scheme includes several major extensions to their technical results. First, we use a different representation for types that integrates polymorphism smoothly with union types and is more computationally efficient. Our representation also supports the incremental definition of new type constructors. Second, an improved check insertion algorithm inserts fewer run-time checks and yields more precise types. Third, our type system addresses the "grubby" features of a real programming language that Cartwright and Fagan's study ignored. In particular, we treat uncurried procedures of fixed and variable arity, assignment, continuations, and top-level definitions. Finally, our system augments Scheme with pattern matching and type definition extensions that facilitate more precise type assignment.

*The first author was supported by a National Defense Science and Engineering Graduate Fellowship.

†The second author was supported by NSF grant CCR-9122518 and Texas Advanced Technology Program grant 003604-014.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

[1] *Safe* implementations of a programming language guarantee to terminate execution with an error message when a primitive is applied to arguments outside its domain

1.1 Outline

The next section illustrates Soft Scheme with an example, describes the type language, and shows the reduction in run-time checking for some bench marks. Section 3 formally defines a soft type system for a functional core language. Section 4 extends this simple soft type system to R4RS Scheme, describes our pattern matching and type definition extensions, and discusses several problems. Sections 5 and 6 discuss related and future work.

2 Soft Scheme

Soft Scheme performs global type checking for Scheme programs. When applied to a program, the type checker writes a version of the program with run-time checks to an output file and displays only a summary of the inserted run-time checks. Type information may then be inspected interactively according to the programmer's interest.

The following program defines and uses a function that flattens a tree to a proper list:[2]

```
(define flatten
  (lambda (l)
    (cond [(null? l) '()]
          [(pair? l) (append (flatten (car l))
                             (flatten (cdr l)))]
          [else (list l)])))
(define a '(1 (2) 3))
(define b (flatten a))
```

Soft type checking this program yields the summary:

```
TOTAL CHECKS    0
```

This program requires no run-time checks as it is completely typable. The types of its top-level definitions follow:

flatten : *(rec ([Y1 (+ nil (cons Y1 Y1) X1)])*
(Y1 –> (list (+ (not cons) (not nil) X1))))
a : *(cons num (cons (cons num nil) (cons num nil)))*
b : *(list num)*

The type of a reflects the shape of the value '(1 (2) 3), which abbreviates the value (cons 1 (cons (cons 2 '()) (cons 3 '()))). Pairs, constructed by cons, have type *(cons · ·)*. The empty list '() has type *nil*. The type for b indicates that b is a proper list of numbers. The type *(list num)* abbreviates *(rec ([Y (+ nil (cons num Y))]) Y)*, which denotes the least fixed point of the recursion equation:

$$Y = nil \cup (cons\ num\ Y).$$

Finally, flatten's type *(Y1 –> (list ...))* indicates that flatten is a procedure of one argument returning a list. The argument type is defined by $Y1 = nil \cup (cons\ Y1\ Y1) \cup X1$ where $X1$ is a type variable standing for any type. Hence, flatten accepts the empty list, pairs, or any other kind of value, *i.e.*, flatten accepts any value. A result returned by flatten is a proper list of elements of type $X1$, which do not include pairs or the empty list.

Now suppose we add the following lines to our program:

[2] A *proper list* is a spine of pairs that ends on the right with the special constant '() called "empty list"

```
(define c (car b))
(define d (map add1 a))
(define e (map sub1 (flatten '(this (that)))))
```

Type checking the extended program yields the summary:

```
c               1 (1 prim)
d               1 (1 prim)
e               1 (1 prim) (1 ERROR)
TOTAL CHECKS    3 (1 ERROR)
```

This extended program requires three run-time checks at primitive operations, one in each of the definitions of c, d, and e. The program output by Soft Scheme shows the locations of the run-time checks:

```
              ⋮

(define c (CHECK-car b))
(define d (map CHECK-add1 a))
(define e (map ERROR-sub1 (flatten '(this (that)))))
```

An unnecessary check at car is inserted because b's type *(list num)* = *(rec ([Y (+ nil (cons num Y))]) Y)* includes *nil* which is not a valid input to car. CHECK-add1 indicates that add1 may fail when applied to some element of a, as indeed it will. Finally, ERROR-sub1 indicates that the occurrence of sub1 in this program never succeeds—if it is ever reached, it will fail. No other run-time checks are required to ensure safe execution of this program. In particular, no run-time checks are required in the body of flatten nor in the bodies of the library routines map and append.

2.1 Presentation Types

Our type checker infers types in an encoded representation that reduces a limited form of subtyping to polymorphism. Since these encoded types are difficult to read, Soft Scheme decodes them into more natural *presentation types* for programmers. Our presentation types can precisely describe a rich collection of subsets of the data domain, yet they are simple for programmers to interpret. Presentation types include *prime types* (P), *tidy union types* (U), and *recursive types* (T).

Conventional Hindley-Milner type systems form types from constants (*e.g. num*, *nil*) and constructors (*e.g. cons*, *–>*) that represent disjoint subsets of the data domain. We call these *prime types* (P):

$$P ::= num \mid nil \mid \ldots \mid (cons\ U_1\ U_2) \mid (U_1 \ldots U_n \text{ –> } U_0)$$

The prime type $(U_1 \ldots U_n \text{ –> } U_0)$ describes procedures that accept n arguments of types $U_1 \ldots U_n$ and return one result of type U_0. We call the constants and constructors forming prime types *tags*, since they correspond to the type tags manipulated by typical implementations of dynamically typed languages.

Our presentation types include a limited form of union type. *Tidy union types* (U) are formed from zero or more prime types and may also include a single type variable (X):

$$U ::= (+\ P_1 \ldots P_n\ [N_1 \ldots N_m\ X]) \mid P \mid X$$

Tidy union types denote the union of the subsets of the data domain that their components denote. Type variables introduce polymorphic types. When a union type includes a type variable, the type variable's range implicitly excludes any types constructed from the tags $P_1 \ldots P_n$ and $N_1 \ldots N_m$ of the union type. For example, in the type *(+ nil (cons num nil) X1)* variable *X1* denotes any type except *nil* or *(cons U_1 U_2)* for any U_1, U_2. A type variable may be preceded by a set of *place holders* (N):

$$N ::= (not\ num) \mid (not\ nil) \mid \ldots \mid (not\ cons) \mid (not\ {->})$$

Place holders further constrain the range of a type variable without broadening the enclosing union type. For instance, flatten's result type *(list (+ (not cons) (not nil) X1))* uses place holders to indicate that the result list does not include pairs or '(). Finally, a union type consisting of only a single prime type is equivalent to that prime type, *i.e.*, $(+\ P) \equiv P$. Similarly, a union type consisting of only a type variable is equivalent to that type variable, *i.e.*, $(+\ X) \equiv X$.

Tidy union types must satisfy two context sensitive properties. First, each tag may be used at most once within a union. This requirement excludes phrases like:

(+ (cons true false) (cons false true))

from our set of types since *cons* is repeated. Second, the same set of tags must precede each occurrence of a particular type variable to ensure that the variable has a consistent range. For example, when *X1* appears with tags *cons* and *nil* in the input type *(+ nil (cons Y1 Y1) X1)* of flatten, it also appears with tags *cons* and *nil* in the output type *(list (+ (not cons) (not nil) X1))*. These properties are invariants required for our type inference algorithm.

The presentation types (T) that are assigned to program identifiers and expressions include recursive types:

$$T ::= (rec\ ([X_1\ U_1] \ldots [X_n\ U_n])\ U_0) \mid U$$

A recursive type denotes the type U_0 where:

$$\begin{aligned} X_1 &= U_1 \\ &\vdots \\ X_n &= U_n \end{aligned}$$

For example, the type *(rec ([Y1 (+ nil (cons X1 Y1))]) Y1)* denotes proper lists containing elements of type *X1*. This type occurs so frequently that we abbreviate it *(list X1)* (that is, *list* is a type macro). By convention, we name type variables bound by recursive types *Yn*.

Following are the types for a few well-known Scheme functions:

map	: *((X1 -> X2) (list X1) -> (list X2))*
member	: *(X1 (list X2) -> (+ false (cons X2 (list X2))))*
read	: *(rec ([Y1 (+ num nil ... (cons Y1 Y1))]) (-> (+ eof num nil ... (cons Y1 Y1))))*
lastpair	: *(rec ([Y1 (+ (cons X1 Y1) X2)]) ((cons X1 Y1) -> (cons X1 (+ (not cons) X2))))*

The higher-order function map takes a function f of type *(X1 -> X2)* and a list x of type *(list X1)*, and applies f to every element of x. It returns a list of the results. Function member takes a key k and a list x, and searches x for an occurrence of k. It returns the first sublist starting with element k if one exists; otherwise, it returns false. Procedure read takes no arguments and parses an "s-expression" from an input device. It returns an *end-of-file* object of type *eof* if no input is available. Finally, lastpair returns the last pair of a non-empty list. Appendix A contains additional examples.

2.2 Performance

Soft Scheme inserts run-time type checks only at uses of primitive operations that it cannot prove safe. Figure 1 summarizes run-time checking for the Scheme versions of the Gabriel Common Lisp bench marks.[3] The percentages

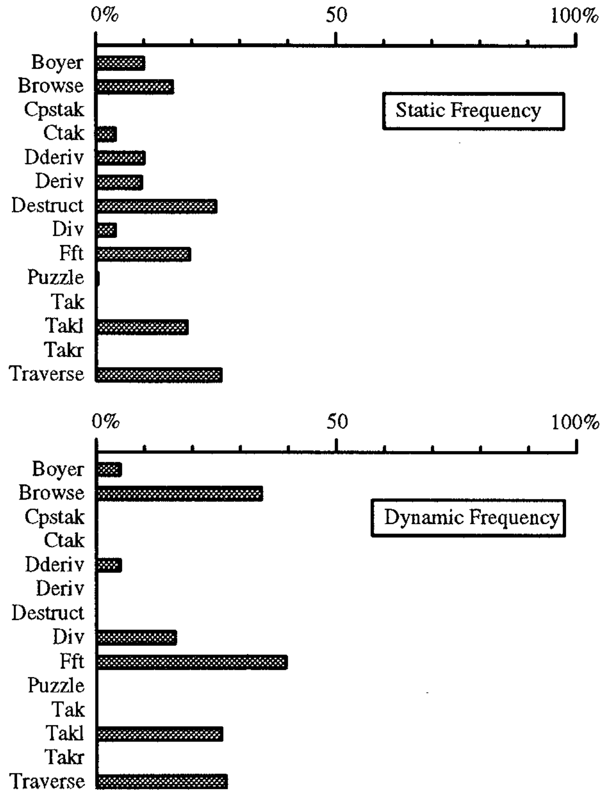

Figure 1: Reduction in Run-Time Checking

indicate how frequently our system inserts run-time checks compared to conventional dynamic typing. The static frequency indicates the incidence of run-time checks inserted in the source code. The dynamic frequency indicates how often the inserted checks are executed. Note that *Cpstak*, *Tak*, and *Takr* are statically typable in our system and hence require no run-time checks at all.

Figure 2 indicates the fraction of total execution time each bench mark spends performing run-time checks under both soft typing and dynamic typing. The timings for dynamic typing were obtained by running the bench marks under Chez Scheme's optimize-level 2 which performs maximum optimization while retaining safety. Since Chez Scheme uses local analysis to eliminate some run-time checks, these programs would spend an even greater fraction of execution time performing run-time checks with a naïve compiler. The timings for soft typing were obtained by running the output from Soft Scheme under Chez's optimize-level 3 which ruthlessly discards run-time checks other than those inserted by Soft Scheme.[4] All measurements were obtained under Chez Scheme 4.1t on an unloaded SparcStation 1.

[3] Obtained from the Scheme Repository at cs.indiana.edu.

[4] Optimize-level 3 still retains argument count checks and some checks in primitives like assoc and member.

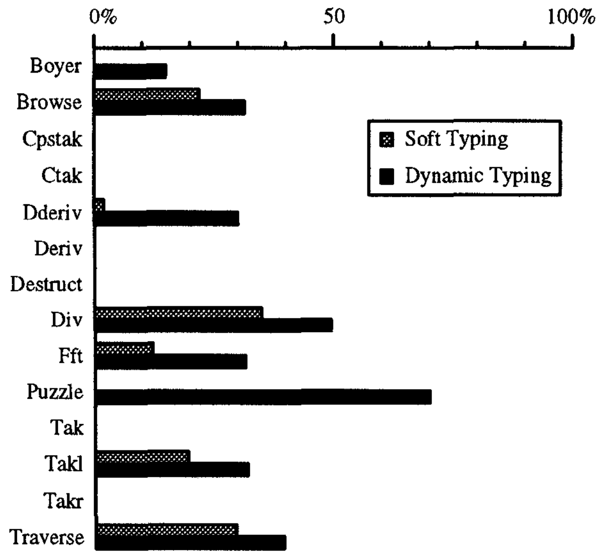

Figure 2: Cost of Run-Time Checking

Soft Scheme significantly decreases run-time checking for all of the bench marks, even though these programs were not written with soft typing in mind. Whether this reduction leads to a significant performance improvement depends on how often the code containing the eliminated checks is executed. Programs like *Browse* and *Traverse*, for example, do not expose enough type information to enable the type checker to remove critical run-time checks. But *Puzzle* illustrates the dramatic benefit (speedup by a factor of 3.3) that can be obtained when run-time type checks are removed from inner loops. As we discuss later, we have developed several extensions to Scheme that facilitate more precise type inference. By using these extensions, we can develop programs that avoid run-time checks where performance is critical. Simple program transformations can further reduce and often completely eliminate the remaining run-time checks.

3 Formal Framework

As the first step in a formal description of our soft type system, we define an idealized, dynamically typed, call-by-value language embodying the essence of Scheme. *Core Scheme* has expressions (e) and values (v) of the following forms:

$$\begin{aligned} e &::= v \mid (\mathbf{ap}\ e_1\ e_2) \mid (\mathbf{CHECK\text{-}ap}\ e_1\ e_2) \mid \\ &\quad\ (\mathbf{if}\ e_1\ e_2\ e_3) \mid (\mathbf{let}\ ([x\ e_1])\ e_2) \\ v &::= c \mid x \mid (\mathbf{lambda}\ (x)\ e) \end{aligned}$$

where $x \in Id$ are *identifiers* and $c \in Const$ are *constants*. *Const* includes *basic constants* (numbers, #t, #f, and '()), *unchecked primitive operations* (add1, car, cons, etc), and *checked primitive operations* (CHECK-add1, CHECK-car, etc). The keywords **ap** and **CHECK-ap** introduce *unchecked* and *checked* applications, which we explain below. *Programs* are closed expressions.

Core Scheme includes both *unchecked* and *checked* versions of every primitive operation. Invalid applications of unchecked primitives, like (**ap** add1 #t) or (**ap** car '()), are meaningless. In an implementation, they can produce arbitrary results ranging from "core dump" to erroneous but apparently valid answers. Checked primitives are observationally equivalent to their corresponding unchecked versions, except that invalid applications of checked primitives terminate execution with an error message. For example, (**ap** CHECK-add1 #t) yields an error message. Similarly, **ap** and **CHECK-ap** introduce unchecked and checked *applications* that are undefined (*resp.* yield an error message) when their first subexpression is not a procedure. Appendix B contains a formal operational semantics for Core Scheme.

Safe implementations of dynamically typed languages like Core Scheme interpret all occurrences of primitive operations as checked operations. Since the run-time checks embedded in checked primitives add overhead to program execution, many implementations allow the programmer to disable run-time type checking—substituting unchecked operations for checked ones. In this mode, valid programs execute faster and give the same answers as they do under conventional "checked" execution, but the language is no longer type safe. Invalid programs can produce arbitrary results ranging from "core dump" to erroneous but apparently valid answers.

3.1 Designing a Soft Type System

Designing a soft type system for Core Scheme is a challenging technical problem. Values in dynamically typed programs belong to many different types, and dynamically typed programs routinely exploit this fact. To accommodate such overlapping types, a soft type system for Core Scheme should include union types and use the following inference rule for applications:

$$\frac{e_1 : (T_1 \text{ -> } T_2) \qquad e_2 : T_3 \qquad T_3 \subseteq T_1}{(\mathbf{ap}\ e_1\ e_2) : T_2}$$

Here $T_3 \subseteq T_1$ indicates that the argument's type must be a subset (or *subtype*) of the function's input union type.

However, conventional Hindley-Milner type systems presume that all monotypes are disjoint. In a Hindley-Milner type system, $T_3 \subseteq T_1$ holds if and only if $T_3 = T_1$. The standard type inference algorithm relies on this fact by using ordinary unification to solve type constraints. Hence the standard algorithm cannot directly accommodate union types. We could base a polymorphic union type system on our presentation types and attempt to find an alternative method of inferring types. Aiken *et al.* [2] have pursued this approach, but its computational complexity is worse than Hindley-Milner typing. We elected instead to modify Hindley-Milner typing to accommodate union types and subtyping without compromising its computational complexity.

To combine union types and subtyping with Hindley-Milner polymorphism, we adapt an encoding Rémy developed for record subtyping [19, 20]. Our encoding permits many union types to be expressed as terms in a free algebra, as with conventional Hindley-Milner types. *Flag variables* enable polymorphism to encode subtyping as subset on union types. Types are inferred by a minor variant of the standard Hindley-Milner algorithm. Soft Scheme translates the inferred types into the presentation types shown above when displaying types to the programmer. Soft Scheme thereby provides the *illusion* of a polymorphic union type system based on presentation types. The illusion is imperfect: occasionally the types generated by Soft Scheme do not match what our informal reasoning leads us to expect. Such a mismatch would be a serious liability for a static type system, as programs would be rejected by the type checker without a clear explanation. In a soft type system, this problem is not nearly as serious. Soft typed programs may contain apparently unmotivated run-time checks, but they can still be executed.

The next two subsections define a collection of static types and a static type inference system based on a variation of Rémy's encoding. The fourth subsection adapts this static type system to a soft type system for Core Scheme. The last subsection indicates how to translate the inferred types into presentation types.

3.2 Static Types

To construct a static type system for Core Scheme, we partition the data domain into disjoint subsets called *prime* types. The partitioning is determined by the domain equation defining the data domain $\mathcal{D}$ for Core Scheme:[5]

$$\mathcal{D} = \mathcal{D}_{num} \oplus \mathcal{D}_{true} \oplus \mathcal{D}_{false} \oplus \mathcal{D}_{nil} \oplus (\mathcal{D} \otimes \mathcal{D}) \oplus [\mathcal{D} \rightarrow_{sc} \mathcal{D}]_{\perp}$$

Each domain constant on the right hand side of the equation identifies a prime type. Similarly, each application of a domain constructor to domains identifies a prime type.

Our static types reflect the partitioning of the data domain. Informally, every static type (σ, τ) is a disjoint union of zero or more prime types $(\kappa^f \vec{\sigma})$ followed by either a single type variable (α) or the empty type (ϕ):

$$\begin{array}{rcl} \sigma, \tau & ::= & \kappa_1^{f_1}\vec{\sigma}_1 \cup \ldots \cup \kappa_n^{f_n}\vec{\sigma}_n \cup \left(\alpha \;\middle|\; \phi\right) \\ f & ::= & \mathbf{+} \mid \mathbf{-} \mid \varphi \end{array}$$

where $\kappa \in Tag = \{num, true, false, nil, cons, \rightarrow\}$. As with presentation types, a type variable's range implicitly excludes types built from the tags $\kappa_1 \ldots \kappa_n$. Each prime type has a *flag* f (written above the tag) that indicates whether the prime type is part of the union type. A flag $\mathbf{+}$ indicates the prime type is present, $\mathbf{-}$ indicates that it is absent, and a flag variable (φ) indicates the prime type may be present or absent depending on how the flag variable is instantiated. Figure 3 presents some examples of types. We use infix notation and write $(\sigma_1 \rightarrow^f \sigma_2) \cup \ldots$ rather than $(\rightarrow^f \sigma_1 \sigma_2) \cup \ldots$ for procedure types. In general, a static type with free flag variables designates a finite set of possible types corresponding to the instances of the free flag variables.

To be well-formed, types must be *tidy*: each tag may be used at most once within a union, and type variables must have a consistent range. Tidiness permits us to find and represent the pairwise unification between two sets of types by performing a single unification step. The following class of grammars defines the tidy types:

$$\begin{array}{l} \sigma^{\emptyset}, \tau^{\emptyset} ::= \alpha^{\emptyset} \mid \phi^{\emptyset} \mid (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^{\emptyset} \cup \tau^{\{\kappa\}} \mid \mu\alpha^{\emptyset}.\tau^{\emptyset} \\ \sigma^X, \tau^X ::= \alpha^X \mid \phi^X \mid (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^X \cup \tau^{X \cup \{\kappa\}} \qquad (\kappa \notin X) \\ f ::= \mathbf{+} \mid \mathbf{-} \mid \varphi \\ \alpha, \beta \in TypeVar \qquad \varphi \in FlagVar \qquad X \in \mathbf{2}^{Tag} \end{array}$$

[5] $\oplus$ denotes the coalesced tagged sum of two Scott domains, $\otimes$ denotes strict Cartesian product, $\rightarrow_{sc}$ is the strict continuous function space constructor, and $[\ ]_{\perp}$ is the lifting construction on domains $\mathcal{D}$ also includes two elements not shown here a check element that is the result of failed run-time checks, and a wrong element that is the meaning of invalid applications like (ap car '()) Every type includes check No type includes wrong.

TypeVar and *FlagVar* are disjoint sets of *type variables* and *flag variables*. Types represent regular trees with the tags κ constructing internal nodes. In a constructed type:

$$(\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^X \cup \tau^{X \cup \{\kappa\}}$$

the constructor κ has arity $n + 2$: flag f, types $\sigma_1 \ldots \sigma_n$, and type τ are its arguments. The parentheses and union symbol $\cup$ are merely syntax to enhance readability. The *labels* X attached to types enforce tidiness by specifying sets of tags that are *unavailable* for use in the type. For example, the phrase:

$$(num^{\mathbf{+}})^{\emptyset} \cup (num^{\mathbf{-}})^{\{num\}} \cup \ldots$$

is not a tidy type because the term $(num^{\mathbf{-}})^{\{num\}}$ violates the restriction $\kappa \notin X$ in the formation of types. The types assigned to program identifiers and expressions have label $\emptyset$. We usually omit labels when writing types. Similarly, an implementation of type inference need not manipulate labels.

Recursive types $\mu\alpha.\tau$ (which must have label $\emptyset$) represent infinite regular trees [3]. The type $\mu\alpha.\tau$ binds α in τ; the usual renaming rules apply to the bound variable α, and we have $\mu\alpha.\tau = \tau[\alpha \mapsto \mu\alpha.\tau]$. Recursive types must be formally contractive, *i.e.*, phrases like $\mu\alpha.\alpha$ are not types. The type $\mu\alpha.\,nil^{\mathbf{+}} \cup (cons^{\mathbf{+}}\,\tau\,\alpha) \cup \phi$, which denotes proper lists of τ, is a common recursive type.

Since our union types denote set-theoretic unions of values, we impose a quotient on types to identify those types that denote the same sets of values. This quotient identifies *i*) types that differ only in the order of union components, and *ii*) types that denote different representations of the empty type:

$$\begin{array}{rcl} \kappa_1^{f_1}\vec{\sigma}_1 \cup \kappa_2^{f_2}\vec{\sigma}_2 \cup \tau & = & \kappa_2^{f_2}\vec{\sigma}_2 \cup \kappa_1^{f_1}\vec{\sigma}_1 \cup \tau \\ \kappa^{\mathbf{-}}\vec{\sigma} \cup \phi & = & \phi. \end{array}$$

It is easy to verify that this quotient preserves tidiness.

To accommodate polymorphism and subtyping, we introduce *type schemes*. A type scheme $\forall\vec{\alpha}\vec{\varphi}.\tau$ is a type with variables $\{\vec{\alpha}\vec{\varphi}\}$ bound in τ. We omit $\forall$ when there are no bound variables, hence types are a subset of type schemes. Type schemes describe sets of types by substitution for bound variables. A *substitution* S is a finite label-respecting[6] map from type variables to types and from flag variables to flags. $S\tau$ (resp. Sf) means the simultaneous replacement of every free variable in the type τ (resp. flag f) by its image under S. A type τ' is an *instance* of type scheme $\forall\vec{\alpha}\vec{\varphi}.\tau$ under substitution S:

$$\tau' \prec_S \forall\vec{\alpha}\vec{\varphi}.\tau$$

if $\text{Dom}(S) = \{\vec{\alpha}\vec{\varphi}\}$ and $S\tau = \tau'$. For example,

$$((num^{\mathbf{+}} \cup \phi) \rightarrow^{\mathbf{+}} (num^{\mathbf{+}} \cup \phi)) \cup \phi$$

is an instance of $\forall\alpha\varphi.(\alpha \rightarrow^{\varphi} \alpha) \cup \phi$ under the substitution $\{\alpha \mapsto (num^{\mathbf{+}} \cup \phi),\ \varphi \mapsto \mathbf{+}\}$.

In our framework, we use polymorphism for both generic types and to express subsets of tidy union types. The type scheme $\forall\varphi_1\varphi_2.\,num^{\varphi_1} \cup nil^{\varphi_2} \cup \phi$ may be instantiated to

[6] Mapping type variables to types with the same label, which preserves tidiness

$num^{+} \cup \emptyset$	means	"numbers;"
$num^{+} \cup nil^{-} \cup \emptyset$	means	"numbers;"
$num^{+} \cup nil^{+} \cup \emptyset$	means	"numbers or '();"
$num^{-} \cup nil^{-} \cup \emptyset$	means	"empty;"
$num^{+} \cup nil^{-} \cup \alpha$	means	"numbers or α but not '();"
$(\alpha \rightarrow^{+} (true^{+} \cup false^{+} \cup \emptyset)) \cup \emptyset$	means	"procedures from α to boolean."

Figure 3: Examples of Types

any type that denotes a subset of $num^{+} \cup nil^{+} \cup \emptyset$. There are four such types:

$$\begin{array}{ll} num^{+} \cup nil^{+} \cup \emptyset & num^{-} \cup nil^{+} \cup \emptyset \\ num^{+} \cup nil^{-} \cup \emptyset & num^{-} \cup nil^{-} \cup \emptyset. \end{array}$$

The informal meaning of these types is explained in Figure 3. By using polymorphic flag variables for the inputs of primitive operations and procedures, we can simulate subtyping at applications while still using unification to equate the procedure's input type and the argument's type. A procedure with type scheme $\forall\varphi_1\varphi_2.(num^{\varphi_1} \cup nil^{\varphi_2} \cup \emptyset) \rightarrow^{+} \ldots$ can be applied to values of types:

$$\begin{array}{l} num^{+} \cup nil^{+} \cup \emptyset, \\ num^{+} \cup nil^{-} \cup \emptyset = num^{+} \cup \emptyset, \\ num^{-} \cup nil^{+} \cup \emptyset = nil^{+} \cup \emptyset, \text{ and} \\ num^{-} \cup nil^{-} \cup \emptyset = \emptyset \end{array}$$

by instantiating the flag variables φ_1 and φ_2 in different combinations of + and −.

Similarly, polymorphic type variables express supersets of types. Basic constants and outputs of primitives may have any type that is a superset of their natural type. For example, numbers have type scheme $\forall\alpha.\, num^{+} \cup \alpha$ which may be instantiated to any type that is a superset of $num^{+} \cup \emptyset$:

$$\begin{array}{l} num^{+} \cup \emptyset \\ num^{+} \cup true^{+} \cup \emptyset \\ num^{+} \cup true^{+} \cup false^{+} \cup \emptyset \\ \vdots \end{array}$$

This ensures that expressions like (if P 1 '()) that mix different types of constants are typable. This expression has type $num^{+} \cup nil^{+} \cup \emptyset$.

The function *TypeOf* maps the constants of Core Scheme to type schemes describing their behavior. The encoding of the unions within a type varies according to whether the union occurs in a negative (input) or positive (output) position. A position is positive if it occurs within the first argument of an even number of $\rightarrow$ constructors, and negative if it occurs within an odd number. With recursive types, a position can be both positive and negative; we assume that primitives do not have such types. For unchecked primitives, negative unions are encoded using variables for valid inputs and − and $\emptyset$ for invalid inputs. Positive unions use + for "present" outputs and variables for "absent" fields. For example, the constants 0, add1, and number? have type schemes:

0	:	$\forall\alpha.\, num^{+} \cup \alpha$
add1	:	$\forall\alpha_1\alpha_2\varphi.((num^{\varphi} \cup \emptyset) \rightarrow^{+} (num^{+} \cup \alpha_1)) \cup \alpha_2$
number?	:	$\forall\alpha_1\alpha_2\alpha_3.(\alpha_1 \rightarrow^{+} (true^{+} \cup false^{+} \cup \alpha_2)) \cup \alpha_3$

The type schemes of checked primitives are similar to those of unchecked primitives, except they never use − or $\emptyset$ since checked primitives accept all inputs. For example, primitive CHECK-add1 has type scheme:

$$\forall\alpha_1\alpha_2\alpha_3\varphi.((num^{\varphi} \cup \alpha_3) \rightarrow^{+} (num^{+} \cup \alpha_1)) \cup \alpha_2.$$

3.3 Static Type Inference

Figure 4 defines a *static type inference* system that assigns types to Core Scheme expressions. Type environments (A) are finite maps from identifiers to type schemes. $A[x \mapsto \tau]$ denotes the functional extension or update of A at x to τ. $FV(\tau)$ returns the free type and flag variables of a type τ. FV extends pointwise to type environments. The *typing* $A \vdash e : \tau$ states that expression e has type τ in type environment A.

Provided the types assigned by *TypeOf* agree with the semantics of Core Scheme, we can prove that this static type system is sound. Appendix B defines a reduction relation $\longmapsto\!\!\!\!\rightarrow$ for which every program either: (i) yields an answer v, (ii) diverges, (iii) yields the error message check, or (iv) gets *stuck* (reaches a non-value normal form, like (ap add1 #t)). *Type soundness* ensures that typable programs yield answers of the expected type and do not get stuck.

Theorem 3.1 (Type Soundness) *If $\emptyset \vdash e : \tau$ then either e diverges, or $e \longmapsto\!\!\!\!\rightarrow$ check, or $e \longmapsto\!\!\!\!\rightarrow v$ and $\emptyset \vdash v : \tau$.*

Proof. We use Wright and Felleisen's technique based on subject reduction [25]. ∎

3.4 Soft Type Checking

The preceding static type system can be used to statically type check Core Scheme programs. The type system will reject programs that contain incorrect uses of unchecked primitives, ensuring safe execution. But the type system will also reject some meaningful programs whose safety it cannot prove. To persuade the type checker to accept an untypable program, a programmer can manually convert it to typable form by judiciously replacing some unchecked operations with checked ones.[7] A soft type checker automates this process.

Figure 5 defines a *soft type inference* system for Core Scheme programs. This system both assigns types and computes a transformed expression in which some unchecked

[7]The same process cannot be used with statically typed languages like ML because the Hindley-Milner type discipline provides insufficient monotypes One must also add explicit definitions of union and recursive types, injections into these types, and projections out of them. The extra injections and projections increase the conceptual complexity of programs and introduce additional run-time overhead.

$$(\textbf{const})\ \frac{\tau \prec_S \mathit{TypeOf}(c)}{A \vdash c : \tau} \qquad (\textbf{var})\ \frac{\tau \prec_S A(x)}{A \vdash x : \tau}$$

$$(\textbf{ap})\ \frac{A \vdash e_1 : (\tau_2 \rightarrow^f \tau_1) \cup \emptyset \qquad A \vdash e_2 : \tau_2}{A \vdash (\textbf{ap}\ e_1\ e_2) : \tau_1} \qquad (\textbf{CHECK-ap})\ \frac{A \vdash e_1 : (\tau_2 \rightarrow^f \tau_1) \cup \tau_3 \qquad A \vdash e_2 : \tau_2}{A \vdash (\textbf{CHECK-ap}\ e_1\ e_2) : \tau_1}$$

$$(\textbf{lam})\ \frac{A[x \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\textbf{lambda}\ (x)\ e) : (\tau_1 \rightarrow^{+} \tau_2) \cup \tau_3} \qquad (\textbf{if})\ \frac{A \vdash e_1 : \tau_1 \qquad A \vdash e_2 : \tau_2 \qquad A \vdash e_3 : \tau_2}{A \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau_2}$$

$$(\textbf{let})\ \frac{A \vdash e_1 : \tau_1 \qquad A[x \mapsto \mathit{Close}(\tau_1, A)] \vdash e_2 : \tau_2}{A \vdash (\textbf{let}\ ([x\ e_1])\ e_2) : \tau_2}$$

$$\begin{array}{rcl} \mathit{Close}(\tau, A) & = & \forall \vec{\alpha}\vec{\varphi}.\, \tau \\ \text{where } \{\vec{\alpha}\vec{\varphi}\} & \subseteq & FV(\tau) - FV(A) \end{array}$$

Figure 4: Static Type Inference

primitives and applications are replaced by checked ones. A *soft typing* $A \mathrel{\models\!\!\!\triangleright} e \Rightarrow e' : \tau$ states that in type environment A, expression e transforms to e' and e' has type τ.

The function *SoftTypeOf* assigns type schemes to constants. For checked primitives and basic constants, *SoftTypeOf* assigns the same type schemes as *TypeOf*. For unchecked primitives, *SoftTypeOf* assigns type schemes that include special *absent variables* ($\tilde{\nu}$). Absent variables record uses of unchecked primitives that may not be safe. Wherever the function *TypeOf* places a $-$ flag or $\emptyset$ type in the input type of a primitive, *SoftTypeOf* places a corresponding absent flag variable $\tilde{\varphi} \in \mathit{AbsFlagVar}$ or absent type variable $\tilde{\alpha} \in \mathit{AbsTypeVar}$. For example, *SoftTypeOf*(add1) is:

$$\forall \alpha_1\alpha_2\tilde{\alpha}_3\tilde{\varphi}.\,((\mathit{num}^{\tilde{\varphi}} \cup \tilde{\alpha}_3) \rightarrow^{+} (\mathit{num}^{+} \cup \alpha_1)) \cup \alpha_2.$$

Absent variables induce classes of *absent flags* ($\tilde{f}$) and *absent types* ($\tilde{\tau}$). Absent flags (resp. types) contain only absent variables:

$$\begin{array}{rcl} \tilde{f} & \in & \{f \mid FV(f) \subset \mathit{AbsFlagVar}\} \\ \tilde{\tau} & \in & \{\tau \mid FV(\tau) \subset (\mathit{AbsFlagVar} \cup \mathit{AbsTypeVar})\}. \end{array}$$

Substitutions are required to map absent flag variables to absent flags and absent type variables to absent types.

If an absent variable is instantiated to a non-empty type in the type assignment process, then the corresponding primitive application must be checked. For example, the expression (**ap** add1 #t) instantiates the absent variable $\tilde{\alpha}_3$ in the type of add1 as (at least) $\mathit{true}^{+} \cup \emptyset$. Since $\mathit{true}^{+} \cup \emptyset$ contains the element true, this application of add1 must be checked. In contrast, the expression (**ap** add1 0) instantiates $\tilde{\alpha}_3$ as $\emptyset$, so no run-time check is necessary. The predicate *empty* used by rules **const-empty** and **ap-empty** in Figure 5 determines whether every member of a set of types and flags is empty. For a single type or flag, *empty* is defined as follows:

$$\begin{array}{rcl} \mathit{empty}(+) & = & \mathit{false} \\ \mathit{empty}(-, \emptyset, \alpha, \tilde{\alpha}, \varphi, \tilde{\varphi}) & = & \mathit{true} \\ \mathit{empty}(\kappa^f \vec{\sigma} \cup \tau) & = & \mathit{empty}(f) \text{ and } \mathit{empty}(\tau) \\ \mathit{empty}(\mu\alpha.\tau) & = & \mathit{empty}(\tau). \end{array}$$

Since a **let**-bound procedure must include run-time checks for all its uses, *absent variables are not generalized by SoftClose.* A simple example demonstrates why this restriction is necessary. The expression:

(let ([inc add1]) (ap inc #t))

requires a run-time check at add1. In typing this expression, add1 is assigned type $((\mathit{num}^{\tilde{\varphi}'} \cup \tilde{\alpha}') \rightarrow^{+} \ldots)$ where $\tilde{\alpha}'$ and $\tilde{\varphi}'$ are fresh variables. Suppose *SoftClose* naïvely generalized absent variables. Generalizing the type of add1 would yield type scheme $\forall \tilde{\alpha}''\tilde{\varphi}''.((\mathit{num}^{\tilde{\varphi}''} \cup \tilde{\alpha}'') \rightarrow^{+} \ldots)$ for inc. In typing the application (**ap** inc #t), $\tilde{\alpha}''$ would be instantiated as $\mathit{true}^{+} \cup \beta$. However, instantiating $\tilde{\alpha}''$ would not affect $\tilde{\alpha}'$ in the type of add1, so no run-time check would be inserted. Section 4.3 describes a better method of inserting checks which performs some extra bookkeeping so that absent variables may be generalized.

Appendix C presents several theorems that establish the correctness of this soft type system.

3.5 Displaying Presentation Types

The types assigned by the soft type inference system are difficult to read. To present more intelligible types to the programmer, we define a translation to the presentation types defined in Section 2.1. This translation eliminates type variables used to encode subtyping and flags.

We define certain type and flag variables as *useless* with respect to a soft typing deduction for the complete program. The definition assumes that all bound variables in the typing differ and are distinct from free variables.

Definition 3.2 (Useless Variable) *With respect to soft typing $\emptyset \mathrel{\models\!\!\!\triangleright} e \Rightarrow e' : \tau$, type or flag variable ν is* useless *if:*

1. *ν is never generalized; or*
2. *ν is an absent variable; or*
3. *ν is generalized in $\forall \nu.\, \tau'$ (that appears in some type environment A in some subdeduction of $\emptyset \mathrel{\models\!\!\!\triangleright} e \Rightarrow e' : \tau$) and ν does not occur negatively in τ'.*

To eliminate variables used to encode subtyping, we replace all useless flag variables with $-$ and all useless type variables with $\emptyset$. To eliminate flags, we replace the remaining flag variables with $+$. As no flag variables remain, displaying presentation types is now a simple matter of translating syntax. Components with flag $+$ translate to ordinary prime types. Components with flag $-$ translate to place holders (*not* κ), or are dropped entirely if the union ends in $\emptyset$.

$$(\textbf{const-empty})\ \frac{\tau \prec_S \mathit{SoftTypeOf}(c) \qquad \mathit{empty}\{S\tilde{\nu} \mid \tilde{\nu} \in \mathrm{Dom}(S)\}}{A \models c \Rightarrow c : \tau}$$

$$(\textbf{const})\ \frac{\tau \prec_S \mathit{SoftTypeOf}(c)}{A \models c \Rightarrow \mathsf{CHECK}\text{-}c : \tau} \qquad (\textbf{var})\ \frac{\tau \prec_S A(x)}{A \models x \Rightarrow x : \tau}$$

$$(\textbf{ap-empty})\ \frac{A \models e_1 \Rightarrow e_1' : (\tau_2 \rightarrow^{f} \tau_1) \cup \tilde{\tau}_3 \qquad A \models e_2 \Rightarrow e_2' : \tau_2 \qquad \mathit{empty}\{\tilde{\tau}_3\}}{A \models (\textbf{ap}\ e_1\ e_2) \Rightarrow (\textbf{ap}\ e_1'\ e_2') : \tau_1}$$

$$(\textbf{ap})\ \frac{A \models e_1 \Rightarrow e_1' : (\tau_2 \rightarrow^{f} \tau_1) \cup \tilde{\tau}_3 \qquad A \models e_2 \Rightarrow e_2' : \tau_2}{A \models (\textbf{ap}\ e_1\ e_2) \Rightarrow (\textbf{CHECK-ap}\ e_1'\ e_2') : \tau_1}$$

$$(\textbf{CHECK-ap})\ \frac{A \models e_1 \Rightarrow e_1' : (\tau_2 \rightarrow^{f} \tau_1) \cup \tau_3 \qquad A \models e_2 \Rightarrow e_2' : \tau_2}{A \models (\textbf{CHECK-ap}\ e_1\ e_2) \Rightarrow (\textbf{CHECK-ap}\ e_1'\ e_2') : \tau_1}$$

$$(\textbf{lam})\ \frac{A[x \mapsto \tau_1] \models e \Rightarrow e' : \tau_2}{A \models (\textbf{lambda}\ (x)\ e) \Rightarrow (\textbf{lambda}\ (x)\ e') : (\tau_1 \rightarrow^{+} \tau_2) \cup \tau_3}$$

$$(\textbf{if})\ \frac{A \models e_1 \Rightarrow e_1' : \tau_1 \qquad A \models e_2 \Rightarrow e_2' : \tau_2 \qquad A \models e_3 \Rightarrow e_3' : \tau_2}{A \models (\textbf{if}\, e_1\ e_2\ e_3) \Rightarrow (\textbf{if}\, e_1'\ e_2'\ e_3') : \tau_2}$$

$$(\textbf{let})\ \frac{A \models e_1 \Rightarrow e_1' : \tau_1 \qquad A[x \mapsto \mathit{SoftClose}(\tau_1, A)] \models e_2 \Rightarrow e_2' : \tau_2}{A \models (\textbf{let}\ ([x\ e_1])\ e_2) \Rightarrow (\textbf{let}\ ([x\ e_1'])\ e_2') : \tau_2}$$

$$\begin{array}{rcl} \mathit{SoftClose}(\tau, A) & = & \forall \vec{\alpha}\vec{\varphi}.\,\tau \\ \text{where } \{\vec{\alpha}\vec{\varphi}\} & \subseteq & FV(\tau) - (FV(A) \cup \mathit{AbsTypeVar} \cup \mathit{AbsFlagVar}) \end{array}$$

Figure 5: Soft Type Inference

To illustrate the translation, consider the following type scheme which approximates that inferred for flatten:

$$\begin{array}{l} \forall \alpha\varphi_1\varphi_2\varphi_3\varphi_4. \\ (\mathit{rec}\ ([y_1\ \ \mathit{nil}^{\varphi_1} \cup (\mathit{cons}^{\varphi_2} y_1 y_1) \cup \alpha] \\ \qquad [y_2\ \ \mathit{nil}^{+} \cup (\mathit{cons}^{+}(\mathit{nil}^{\varphi_3} \cup (\mathit{cons}^{\varphi_4} y_1 y_1) \cup \alpha) y_2) \cup \tilde{\alpha}_2]) \\ \quad y_1 \rightarrow^{+} y_2 \cup \tilde{\alpha}_3) \end{array}$$

Variables $\tilde{\alpha}_2, \tilde{\alpha}_3, \varphi_3, \varphi_4$ are useless: $\tilde{\alpha}_2$ and $\tilde{\alpha}_3$ because they are absent variables, and φ_3 and φ_4 because they do not occur negatively in the above type. Replacing the useless variables with $\emptyset$ and $-$ as appropriate, and replacing the remaining flag variables φ_1, φ_2 with $+$ yields:

$$\begin{array}{l} \forall \alpha. \\ (\mathit{rec}\ ([y_1\ \ \mathit{nil}^{+} \cup (\mathit{cons}^{+} y_1 y_1) \cup \alpha] \\ \qquad [y_2\ \ \mathit{nil}^{+} \cup (\mathit{cons}^{+}(\mathit{nil}^{-} \cup (\mathit{cons}^{-} y_1 y_1) \cup \alpha) y_2) \cup \emptyset]) \\ \quad y_1 \rightarrow^{+} y_2 \cup \emptyset) \end{array}$$

Changing syntax, we have:

$$\begin{array}{l} \forall \alpha. \\ \mathit{(rec\ ([y_1\ \ (+\ nil\ (cons\ y_1\ y_1)\ \alpha)]} \\ \qquad \mathit{[y_2\ \ (+\ nil\ (cons\ (+\ (not\ nil)\ (not\ cons)\ \alpha)\ y_2))]])} \\ \quad \mathit{(y_1 \to y_2))} \end{array}$$

A presentation type resulting from this translation may not completely capture all of the information present in the internal representation. When the internal type has a flag variable that appears in both positive and negative positions, the input-output dependence encoded by this flag variable is lost. For example, in type scheme:

$$\forall \varphi_1\varphi_2.(\mathit{true}^{\varphi_1} \cup \mathit{false}^{\varphi_2} \cup \emptyset) \rightarrow^{+} (\mathit{true}^{\varphi_1} \cup \mathit{false}^{\varphi_2} \cup \emptyset) \cup \emptyset$$

flag variable φ_1 (resp. φ_2) indicates that this function returns true (resp. false) only if it is passed true (resp. false). This dependence is lost in translating to the presentation type *((+ true false) –> (+ true false)).* This is a source of imperfection in our illusion of a polymorphic union type system based on presentation types (discussed in Section 3.1). This imperfection does not seem to matter for practical programming.

4 Practical Implementation

A practical soft type system must address the features of a real programming language. This section extends our simple soft type system to R4RS Scheme, and presents two extensions to Scheme that enable more precise type inference. We also discuss several problems.

4.1 Typing Scheme

Scheme procedures may have a fixed arity or accept an unlimited number of arguments. Certain primitives also accept trailing optional arguments. We encode procedure types with the binary constructor –>∗ whose first argument is an *argument list.* Argument lists are encoded by the binary constructor *arg* and the constant *noarg.* Hence the type *(X1 X2 X3 –> X4)* merely abbreviates:

((arg X1 (arg X2 (arg X3 noarg))) –>∗ X4).

The types of procedures of unlimited arity use recursive types; for example:

$+$: *(rec ([Y1 (+ noarg (arg num Y1))]) (Y1 –> num)).*

A consequence of this encoding is that run-time checks caused by applying procedures to the wrong number of arguments are distinguished from other run-time checks. In practice, we find that such argument count checks usually indicate program errors.

Assignment and the continuation operator call/cc are important features of Scheme. There are several solutions to typing assignment and continuations in a polymorphic framework. Our prototype uses the simplest method which restricts polymorphism to **let**-expressions where the bound expression is a syntactic value [24]. For Scheme, all expressions are values except those that contain an application of a non-primitive procedure or an "impure" primitive (like cons or call/cc).

In our prototype, call/cc has the type:

(((X1 -> X2) -> X1) -> X1).

A use of call/cc may require a run-time check for either of two reasons: (*i*) the value to which call/cc is applied (of type *((X1 -> X2) -> X1))* is not a procedure of one argument; or (*ii*) the continuation obtained (of type *(X1 -> X2))* is not treated as a procedure of one argument. While the first case could be handled as usual by inserting CHECK-call/cc, the second case cannot. To address the second case, we replace each occurrence of call/cc in the program with the expression:

```
(lambda (v)
  (call/cc (lambda (k) (v (lambda (x) (k x))))))
```

This transformation, a composition of three η-expansions, introduces an explicit **lambda**-expression for the continuation. The expression (**lambda** (x) (k x)) will be checked if the continuation may be mistreated.

A Scheme program is a sequence of definitions that may refer forwards or backwards to other definitions. Treating an entire program as a single **letrec**-expression is not satisfactory because the Hindley-Milner type system assigns polymorphic types to identifiers only within the body of a **let**- or **letrec**-expression, not within the bindings. To obtain polymorphism for definitions, we topologically sort the program's definitions into strongly connected components. The components form a tree that may be organized as nested **letrec**-expressions and typed in the usual manner.

Scheme's **letrec**-expression is the only significant obstacle to typing R4RS Scheme. In a **letrec**-expression:

(**letrec** ([x_1 e_1] ... [x_n e_n]) e)

the bindings $e_1 \ldots e_n$ may be arbitrary expressions. They are evaluated in some unspecified order. An expression like (**letrec** ([x (**not** x)]) x) that refers to the value of some x_i before e_i has been evaluated is invalid. But R4RS Scheme implementations are not required to detect such invalid expressions. A conforming implementation may instead return an unspecified value. To ensure that the value returned by any **letrec**-expression is an element of the expression's type (*i.e.* that the type system is sound), the Scheme implementation must detect invalid **letrec**-expressions.

4.2 Extensions to Scheme

Our prototype includes two natural extensions to Scheme that enable more precise type assignment. Pattern matching enables the type checker to "learn" from type tests. For example, in the expression:

```
(let ([x (if P 0 (cons 1 '()))])
  (if (pair? x) (car x) 2))
```

no check should be necessary at car. However, as our type system assigns types to identifiers, the occurrence of x in (car x) has the same type as every other occurrence of x. This type includes *num*, hence a run-time check is inserted. In contrast, the equivalent code:[8]

```
(let ([x (if P 0 (cons 1 '()))])
  (match x [(a . _) a] [_ 2]))
```

couples the type test to the decomposition of x. By extending the type system to directly type pattern matching expressions, we avoid the unnecessary run-time check. To improve the treatment of ordinary Scheme programs that do not use pattern matching, we translate simple forms of type testing **if**-expressions, like that above, into equivalent **match**-expressions.

Our second extension to Scheme is a type definition facility that allows the introduction of new type constructors. The expression:

(**define-structure** (Foo a b c))

defines constructors, predicates, selectors, and mutators for data of type *(Foo $\cdots$)*. Programs that use type definitions are assigned more informative and more precise types than those that encode data structures using lists or vectors. A similar facility defines immutable data.

4.3 Problems

We have identified three problems with our system that result in imprecise typing.

Tidiness: Our tidy union types can express most common Scheme types. However, no decidable type system can express all computable subsets of the data domain. Four of the R4RS Scheme procedures do not have a tidy union type. These are: map and for-each for an arbitrary number of arguments; apply with more than two arguments; and append when the last element is not a list. The types inferred for these procedures are too coarse. Some uses of these procedures require run-time checks. Overall, we feel that our type language provides a good balance between simplicity and expressiveness.

Reverse Flow: Several typing rules require that the types of two subexpressions be identical. For instance, **if**-expressions require their then- and else-clauses to have the same type. Applications of non-polymorphic procedures require the types of arguments to match the types the function expects (see rules **ap** and **Cap** in Figure 4). Consequently, type information flows both with and counter to the direction of value flow. Reverse flow can cause an inaccurate type to be inferred even though a more accurate tidy union type exists. For example, the following function:

(**define** f (**lambda** (x) (**if** *P* x #f)))

[8] The expression (**match** *e* [pat_1 e_1] .. [pat_n e_n]) compares the value of *e* against *patterns* pat_1 . . pat_n. Any identifiers in the first matching pattern pat_i are bound to corresponding parts of the value of *e*, and e_i is evaluated in the extended environment The pattern (pat_1 pat_2) matches a pair whose components match pat_1 and pat_2 Pattern _ matches anything.

is inferred type $((\mathit{false}^{+} \cup \alpha_1) \rightarrow^{+} (\mathit{false}^{+} \cup \alpha_1)) \cup \alpha_2$. The constant #f forces false^{+} into the type of x, and therefore into the input type of f. Hence the subtyping provided by polymorphism fails at applications of f—all arguments to f are forced to include false^{+} in their type. Were $((\mathit{false}^{\varphi} \cup \alpha_1) \rightarrow^{+} (\mathit{false}^{+} \cup \alpha_1)) \cup \alpha_2$ inferred for f, subtyping would work at uses of f.

The method of inserting run-time checks described in subsection 3.4 exacerbates the reverse flow problem. The insertion of a run-time check can cascade, forcing the insertion of many other unnecessary run-time checks. For example, the following program requires no run-time checks:

```
(let ([f add1]) (lambda (x) (f x) (* 2 x)))
```

In this program, f has type scheme $\forall\varphi\alpha_2\alpha_3.\,((\mathit{num}^{\varphi} \cup \tilde{\alpha}) \rightarrow^{+} (\mathit{num}^{+} \cup \alpha_2)) \cup \alpha_3$. Suppose we add the application (f #t) between (f x) and (* 2 x). Now the absent variable $\tilde{\alpha}$ that is not generalized by the let-expression is replaced with $\mathit{true}^{+} \cup \tilde{\alpha}'$, and a run-time check is inserted at add1. But the input type of f is now $\mathit{num}^{\varphi} \cup \mathit{true}^{+} \cup \tilde{\alpha}'$, hence reverse flow at the application (f x) forces the type of x to include true^{+}. Therefore * receives an unnecessary run-time check.

Our prototype avoids cascading by using a better technique to insert run-time checks. Absent variables are generalized by **let**-expressions in the same manner as ordinary variables. But whenever a generalized absent variable is instantiated, the instance type is recorded. A primitive requires a run-time check if any of the instance types of its absent variables are non-empty. An instance type is non-empty if it contains a +-flag or if any of its instances are non-empty. The extra bookkeeping required for this technique is minimal. The improvement in typing precision and the attendant reduction in run-time checking can be significant.

We have also investigated several adaptations of structural subtyping [12, 17] to address the reverse flow problem. Structural subtyping is more powerful than encoding subtyping with polymorphism as it permits subtyping at all function applications. By permitting more subtyping, soft type systems based on structural subtyping can infer more precise types. However, our experience to date with such systems has been disappointing. The inference algorithms we have constructed for structural subtyping with union and recursive types require exorbitant amounts of memory for even small examples.

Assignment: Because assignment interferes with polymorphism, and therefore with subtyping, assignment can be a major source of imprecision. Scheme includes both assignable identifiers, set by **set!**, and assignable pairs, set by set-car! and set-cdr!. Assignments to local identifiers seldom cause trouble. However, assignments to global identifiers or to pairs disable subtyping, and hence may cause the accumulation of large, inaccurate types. Using immutable pairs when possible adequately addresses the problem for set-car! and set-cdr!. At present, we have no satisfactory solutions for global identifier assignments.

5 Related Work

Our practical soft type system is based on a soft type system designed by Cartwright and Fagan for an idealized functional language [5, 7]. Cartwright and Fagan discovered how to incorporate a limited form of union type in a Hindley-Milner polymorphic type system. Their method is based on an encoding technique Rémy developed to reduce record subtyping to polymorphism [19]. Their system represents union types in a different manner from ours, but their types can be viewed as the types of Section 3.2 with all type variables having label $\emptyset$. This precludes type variables from appearing in unions. A type like:

$$((\mathit{false}^{+} \cup \alpha) \rightarrow^{+} (\mathit{false}^{-} \cup \alpha)) \cup \phi$$

where α has label $\{\mathit{false}\}$ must instead be represented by enumerating all other tags in place of α:

$$(\mathit{false}^{+} \cup \mathit{num}^{\varphi_1} \cup \ldots \cup (\mathit{cons}^{\varphi_n}\ \alpha_1\ \alpha_2)) \rightarrow^{+}$$
$$(\mathit{false}^{-} \cup \mathit{num}^{\varphi_1} \cup \ldots \cup (\mathit{cons}^{\varphi_n}\ \alpha_1\ \alpha_2))$$

In Carwright and Fagan's system, procedures like flatten from Section 2 have large types that do not have a natural decoding into presentation types. Furthermore, their representation does not support incremental definition of new type constructors, and type inference is not particularly efficient because simple types can have large representations.

Aiken and Wimmers have recently developed a sophisticated soft type system for the functional language FL [1, 2]. Their system supports a rich type language including union types, recursive types, intersection types, conditional types, and subtype constraints. While it seems clear that their formal system assigns more precise types to some programs than our system does, their implementation discards some solutions for the sake of efficiency. Consequently, their implementation can yield less precise types than ours for some simple programs. Even with this concession to efficiency, both their timing results and the complexity of their algorithm indicate that it is slower than ours. And the inferred types are probably too complicated to be easily interpreted by programmers. Nevertheless, if their system can be extended to include imperative features (assignment and control) and acceptable performance can be achieved, we believe that it could serve as a good basis for a stronger soft type system for Scheme.

Several researchers have developed static type systems that extend the Hindley-Milner type discipline by adding a maximal type $\top$ as the type of otherwise untypable phrases [8, 10, 18, 22, 23]. This framework is too imprecise to form the basis for a soft type system because it does not support union types or inferred recursive types. The frequency with which $\top$ is assigned as the type of an expression limits the utility of the inferred type information. Nevertheless, Henglein has used a formulation of static typing enhanced with $\top$ to eliminate many run-time checks from Scheme programs.

The designers of optimizing compilers for Scheme and Lisp have developed type analyzers based on data flow analysis [4, 9, 13, 14, 15, 21]. The information gathered by these systems is important for program optimization, but it is much too coarse to serve as the basis for a soft type system. None of the systems infer polymorphic types and most infer types that are simple unions of type constants and constructions.

6 Future Work

Our current implementation processes an entire program at once, inferring type information and inserting run-time checks throughout. As such, the system is not well suited to large scale software development. We are investigating module systems to enable separate type checking and compilation of different parts of a program.

Announcement

Soft Scheme is available by anonymous FTP from cs.rice.edu in file public/wright/soft.tar.Z. Our pattern matching and type definition extensions for Scheme, which may be used independently of Soft Scheme, are also available from cs.rice.edu in file public/wright/match.tar.Z.

Acknowledgements

Kent Dybvig extended Chez Scheme overnight to permit mixing checked and unchecked primitives within one procedure. Without his assistance, we could not have obtained realistic execution time measurements for soft typed programs.

Appendices

A Examples

Following are some simple functions and their inferred types. None of these functions require run-time checks if they are passed arguments within their intended domain.

```
(define map     ; apply a function to every element of a list
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l)) (map f (cdr l))))))
;; ((X1 -> X2) (list X1) -> (list X2))

(define member     ; search for a key in a list
  (lambda (x l)
    (match l
      [() #f]
      [(y . rest) (if (equal? x y)
                      l
                      (member x rest))])))
;; (X1 (list X2) -> (+ false (cons X2 (list X2))))

(define lastpair     ; find the last pair of a non-empty list
  (lambda (s)
    (if (pair? (cdr s))
        (lastpair (cdr s))
        s)))
;; (rec ([Y1 (+ (cons X1 Y1) X2)])
;;   ((cons X1 Y1) -> (cons X1 (+ (not cons) X2))))

(define subst*   ; substitution for trees
  (lambda (new old t)
    (cond [(eq? old t) new]
          [(pair? t) (cons (subst* new old (car t))
                           (subst* new old (cdr t)))]
          [else t])))
;; (rec ([Y1 (+ (cons Y1 Y1) X1)])
;;   (Y1 X2 Y1 -> Y1))

(define append
  (lambda l
    (cond [(null? l) ()]
          [(null? (cdr l)) (car l)]
          [else (let loop ([m (car l)])
                  (if (null? m)
                      (apply append (cdr l))
                      (cons (car m) (loop (cdr m)))))])))
;; ((arglist (list X1)) ->* (list X1))

(define taut?      ; test for a tautology
  (lambda (x)
    (match x
      [#t #t]
      [#f #f]
      [_ (and (taut? (x #t)) (taut? (x #f)))])))
;; (rec ([Y1 (+ false true ((+ false true) -> Y1))])
;;   (Y1 -> (+ false true)))

;; from Aiken and Wimmers [2]
(define Y      ; least fixed point combinator
  (lambda (f)
    (lambda (y)
      (((lambda (x) (f (lambda (z) ((x x) z))))
        (lambda (x) (f (lambda (z) ((x x) z)))))
       y))))
;; (((X1 -> X2) -> (X1 -> X2)) -> (X1 -> X2))
(define last    ; find last element of a list
  (Y (lambda (f)
       (lambda (x)
         (if (null? (cdr x))
             (car x)
             (f (cdr x)))))))
;; ((cons Z1 (list Z1)) -> Z1)
```

B Operational Semantics

Figure 6 specifies a reduction semantics for Core Scheme (neglecting pairs, which are easy to add). *Val* is the set of values. *Prim* $\subset$ *Const* is the set of primitive operations. Answers are values or the special token check which is returned by programs that apply checked operations to invalid arguments. The reduction relation depends on a definition of *evaluation contexts*, E. An evaluation context is an expression with one subexpression replaced by a hole, $[]$. $E[e]$ is the expression obtained by placing e in the hole of E. Our definition of evaluation contexts ensures that applications evaluate from left to right,[9] as every non-value expression can be uniquely decomposed into an evaluation context and a redex.

Rule *let* reduces **let**-expressions by substitution. Rules β_v and *check*-β_v reduce ordinary and checked applications of **lambda**-expressions. Rules δ_1, δ_2, *check*-δ_1, and *check*-δ_2 use the partial function:

$$\delta : Prim \times ClosedVal \rightharpoonup (ClosedVal \cup \{\text{check}\})$$

to interpret the application of primitives. *ClosedVal* is the set of closed values. For all checked primitives CHECK-c, we require that $\delta(\text{CHECK-}c, v)$ be defined for all closed values v. For unchecked primitives, δ may be undefined at some arguments. For corresponding pairs of unchecked and

[9] Our theorems also hold for a language that does not specify the evaluation order, like Scheme.

$$\begin{array}{lrcll}
(\beta_v) & E[(\textbf{ap}\ (\textbf{lambda}\ (x)\ e)\ v)] & \longmapsto & E[e[x \mapsto v]] & \\
(\delta_1) & E[(\textbf{ap}\ c\ v)] & \longmapsto & E[\delta(c, v)] & \text{if } c \in \mathit{Prim} \text{ and } \delta(c, v) \in \mathit{Val} \\
(\delta_2) & E[(\textbf{ap}\ c\ v)] & \longmapsto & \mathsf{check} & \text{if } c \in \mathit{Prim} \text{ and } \delta(c, v) = \mathsf{check} \\
(\mathit{check}\text{-}\beta_v) & E[(\textbf{CHECK-ap}\ (\textbf{lambda}\ (x)\ e)\ v)] & \longmapsto & E[e[x \mapsto v]] & \\
(\mathit{check}\text{-}\delta_1) & E[(\textbf{CHECK-ap}\ c\ v)] & \longmapsto & E[\delta(c, v)] & \text{if } c \in \mathit{Prim} \text{ and } \delta(c, v) \in \mathit{Val} \\
(\mathit{check}\text{-}\delta_2) & E[(\textbf{CHECK-ap}\ c\ v)] & \longmapsto & \mathsf{check} & \text{if } c \notin \mathit{Prim} \text{ or } \delta(c, v) = \mathsf{check} \\
(\mathit{if}_1) & E[(\textbf{if}\ v\ e_1\ e_2)] & \longmapsto & E[e_1] & \text{if } v \neq \#\mathsf{f} \\
(\mathit{if}_2) & E[(\textbf{if}\ \#\mathsf{f}\ e_1\ e_2)] & \longmapsto & E[e_2] & \\
(\mathit{let}) & E[(\textbf{let}\ ([x\ v])\ e)] & \longmapsto & E[e[x \mapsto v]] &
\end{array}$$

$$E \ ::= \ [] \mid (\textbf{ap}\ E\ e) \mid (\textbf{ap}\ v\ E) \mid (\textbf{CHECK-ap}\ E\ e) \mid (\textbf{CHECK-ap}\ v\ E) \mid (\textbf{if}\ E\ e_1\ e_2) \mid (\textbf{let}\ ([x\ E])\ e)$$

Figure 6: Reduction Semantics for Core Scheme

checked primitives (c, CHECK-c), we require that $\delta(c, v)$ and $\delta(\text{CHECK-}c, v)$ agree for all v, except that $\delta(c, v)$ may be undefined when $\delta(\text{CHECK-}c, v)$ yields check:

$$\delta(\text{CHECK-}c, v) = \begin{cases} \delta(c, v) & \text{if } \delta(c, v) \text{ is defined;} \\ \mathsf{check} & \text{if } \delta(c, v) \text{ is undefined.} \end{cases}$$

When δ returns check for the application of a primitive, check immediately becomes the program's answer via rule δ_2 or *check*-δ_2. Rule *check*-δ_2 also ensures that checked applications of basic constants, like (**CHECK-ap** 1 2), result in answer check.

Let $\longmapsto\!\!\!\!\rightarrow$ be the reflexive and transitive closure of $\longmapsto$. With unchecked operations, evaluation can lead to a normal form (relative to $\longmapsto\!\!\!\!\rightarrow$) that is neither a value nor check. Such normal forms arise when an unchecked primitive is applied to an argument for which it is not defined, *e.g.*, (**ap** add1 #t), or when the first subexpression of an unchecked application is not a procedure, *e.g.*, (**ap** 1 2). We say such an expression is ***stuck***. Say that e diverges when there is an infinite reduction sequence $e \longmapsto e' \longmapsto e'' \longmapsto \ldots$ All closed expressions either (*i*) yield an answer that is a closed value, (*ii*) diverge, (*iii*) yield check, or (*iv*) become stuck.

Lemma B.1 *For closed expressions e, either $e \longmapsto\!\!\!\!\rightarrow v$ where v is closed, e diverges, $e \longmapsto\!\!\!\!\rightarrow$ check, or $e \longmapsto\!\!\!\!\rightarrow e'$ where e' is stuck.*

C Correctness

Three theorems establish the correctness of our soft type system. The first states that all programs can be soft typed.

Theorem C.1 (Applicability) *For all programs e, there exist e' and τ such that $\emptyset \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$.*

Proof. The proof proceeds by induction over the structure of typing derivations, using a strengthened induction hypothesis to accommodate terms with free identifiers. ∎

The second theorem establishes that soft typed programs do not become stuck.

Theorem C.2 (Soft Soundness) *If $\emptyset \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$ then either e' diverges, or $e' \longmapsto\!\!\!\!\rightarrow$ check, or $e' \longmapsto\!\!\!\!\rightarrow v'$.*

Proof. To establish this theorem, we first show that e' has a type in the static type system. Let τ^* be the result of replacing all absent flag variables with $-$ and all absent type variables with $\emptyset$ in τ. Define A^* similarly, and where a type scheme binds an absent variable, eliminate the binding.

Lemma C.3 *If $A \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$ then $A^* \vdash e' : \tau^*$.*

This lemma is proved by induction over the structure of the deduction $A \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$. The theorem then follows by Theorem 3.1 (Type Soundness). ∎

To prove that the soft typed program and the original program are equivalent, recall that evaluation has four possible outcomes. A program may (*i*) yield an answer v, (*ii*) diverge, (*iii*) yield check, or (*iv*) get stuck. Let $e \sqsubseteq e'$ mean that e' may have more checked operations that e, but e and e' are otherwise the same. Specifically, $\sqsubseteq$ is the reflexive, transitive, and compatible[10] closure of the following relation:

$$c \sqsubseteq_0 \text{CHECK-}c \qquad \frac{e_1 \sqsubseteq_0 e_1' \qquad e_2 \sqsubseteq_0 e_2'}{(\textbf{ap}\ e_1\ e_2) \sqsubseteq_0 (\textbf{CHECK-ap}\ e_1'\ e_2')}$$

Soft type checking preserves the meaning of programs, but lifts the meaning of invalid programs that get stuck to check.

Theorem C.4 (Correspondence) *If $\emptyset \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$ then:*

$$\begin{array}{lcll}
e \longmapsto\!\!\!\!\rightarrow v & \Leftrightarrow & e' \longmapsto\!\!\!\!\rightarrow v' & \textit{where } v \sqsubseteq v'; \\
e \textit{ diverges} & \Leftrightarrow & e' \textit{ diverges}; & \\
e \longmapsto\!\!\!\!\rightarrow \mathsf{check} \textit{ or } e \textit{ gets stuck} & \Leftrightarrow & e' \longmapsto\!\!\!\!\rightarrow \mathsf{check}. &
\end{array}$$

Proof. We first show that a program that has fewer checked operations performs the same evaluation steps, but may become stuck sooner.

Lemma C.5 (Simulation) *For $e_1 \sqsubseteq e_1'$:*

$$\begin{array}{llcl}
\textit{1.} & e_1 \longmapsto e_2 & \Rightarrow & e_1' \longmapsto e_2' \textit{ and } e_2 \sqsubseteq e_2'; \\
& e_1 \longmapsto \mathsf{check} & \Rightarrow & e_1' \longmapsto \mathsf{check}; \\
& e_1 \textit{ is stuck} & \Rightarrow & e_1' \longmapsto \mathsf{check} \textit{ or } e_1' \textit{ is stuck.} \\
\textit{2.} & e_1' \longmapsto e_2' & \Rightarrow & e_1 \longmapsto e_2 \textit{ and } e_2 \sqsubseteq e_2'; \\
& e_1' \longmapsto \mathsf{check} & \Rightarrow & e_1 \longmapsto \mathsf{check} \textit{ or } e_1 \textit{ is stuck.}
\end{array}$$

Both parts of this lemma are proved by case analysis on the structure of the expressions.

For the first part, from $\emptyset \mathrel{|\!\!\!\gg} e \Rightarrow e' : \tau$ we have $e \sqsubseteq e'$ by induction and case analysis of the soft typing rules in

[10]The *compatible* closure of a relation R is $\{(C[e_1], C[e_2])\}$ for all $(e_1, e_2) \in R$ and all contexts C. A context C is an expression with a hole in place of one subexpression.

Figure 5. By induction with the first part of Simulation:

$$\begin{array}{lcl} e \longmapsto\!\!\!\!\rightarrow v & \Rightarrow & e' \longmapsto\!\!\!\!\rightarrow v' \textit{ and } v \sqsubseteq v'; \\ e \textit{ diverges} & \Rightarrow & e' \textit{ diverges}; \\ e \longmapsto\!\!\!\!\rightarrow \mathsf{check} & \Rightarrow & e' \longmapsto\!\!\!\!\rightarrow \mathsf{check}; \\ e \textit{ gets stuck} & \Rightarrow & e' \longmapsto\!\!\!\!\rightarrow \mathsf{check} \textit{ or } e' \textit{ gets stuck}. \end{array}$$

But by Soft Soundness e' cannot get stuck.

For the second part, again $e \sqsubseteq e'$. By induction with the second part of Simulation:

$$\begin{array}{lcl} e' \longmapsto\!\!\!\!\rightarrow v' & \Rightarrow & e \longmapsto\!\!\!\!\rightarrow v \textit{ and } v \sqsubseteq v'; \\ e' \textit{ diverges} & \Rightarrow & e \textit{ diverges}; \\ e' \longmapsto\!\!\!\!\rightarrow \mathsf{check} & \Rightarrow & e \longmapsto\!\!\!\!\rightarrow \mathsf{check} \textit{ or } e \textit{ gets stuck}. \end{array}$$

This completes the proof. ∎

References

[1] Aiken, A., and Wimmers, E. L. Type inclusion constraints and type inference. *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture* (1993), 31–41.

[2] Aiken, A., Wimmers, E. L., and Lakshman, T. K. Soft typing with conditional types. *Proceedings of the 21st Annual Symposium on Principles of Programming Languages* (January 1994), 163–173.

[3] Amadio, R. M., and Cardelli, L. Subtyping recursive types. *Proceedings of the 17th Annual Symposium on Principles of Programming Languages* (January 1990), 104–118.

[4] Beer, R. D. Preliminary report on a practical type inference system for Common Lisp. *Lisp Pointers 1*, 2 (1987), 5–11.

[5] Cartwright, R., and Fagan, M. Soft typing. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), 278–292.

[6] Clinger, W., Rees, J., et al. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers IV* (July-September 1991).

[7] Fagan, M. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* PhD thesis, Rice University, October 1990.

[8] Gomard, C. K. Partial type inference for untyped functional programs. *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (June 1990), 282–287.

[9] Heintze, N. Set based analysis of ML programs. Tech. Rep. CMU-CS-93-193, Carnegie Mellon University, July 1993.

[10] Henglein, F. Global tagging optimization by type inference. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (June 1992), 205–215.

[11] Hindley, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (December 1969), 29–60.

[12] Kaes, S. Type inference in the presence of overloading, subtyping and recursive types. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (June 1992), 193–204.

[13] Kaplan, M. A., and Ullman, J. D. A scheme for the automatic inference of variable types. *Journal of the Association for Computing Machinery 27*, 1 (January 1980), 128–145.

[14] Kind, A., and Friedrich, H. A practical approach to type inference in EuLisp. *Lisp and Symbolic Computation 6*, 1/2 (August 1993), 159–175.

[15] Ma, K. L., and Kessler, R. R. TICL—a type inference system for Common Lisp. *Software Practice and Experience 20*, 6 (June 1990), 593–623.

[16] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[17] Mitchell, J. C. Type inference with simple subtypes. *Journal of Functional Programming 1*, 3 (July 1991), 245–286. Preliminary version in: Coercion and Type Inference, *Proc. 11th Annual Symposium on Principles of Programming Languages*, 1984, pp. 175–185.

[18] O'Keefe, P. M., and Wand, M. Type inference for partial types is decidable. In *Proceedings of the European Symposium on Programming, LNCS 582* (1992), Springer-Verlag, pp. 408–417.

[19] Rémy, D. Typechecking records and variants in a natural extension of ML. *Proceedings of the 16th Annual Symposium on Principles of Programming Languages* (January 1989), 77–87.

[20] Rémy, D. Type inference for records in a natural extension of ML. Tech. Rep. 1431, INRIA, May 1991.

[21] Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991. Also: Tech. Rep. CMU-CS-91-145.

[22] Thatte, S. R. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium, LNCS 317* (July 1988), Springer-Verlag, pp. 615–629.

[23] Thatte, S. R. Quasi-static typing. *Proceedings of the 17th Annual Symposium on Principles of Programming Languages* (January 1990), 367–381.

[24] Wright, A. K. Polymorphism for imperative languages without imperative types. Tech. Rep. 93-200, Rice University, February 1993.

[25] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 91-160, Rice University, April 1991. To appear in: *Information and Computation*, 1994.