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.

Rice University Technical Report TR93-218

December 6, 1993

Abstract

Soft type systems provide the benefits of static type checking for dynamically typed languages without rejecting untypable programs. A soft type checker infers types for variables and expressions and inserts explicit run-time checks to transform untypable programs to typable form. We describe a practical soft type system for R4RS Scheme. Our 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 procedures of fixed and variable arity, assignment, continuations, and top-level definitions. Our implementation is available by anonymous FTP.

---

*The first author was supported in part by the United States Department of Defense under a National Defense Science and Engineering Graduate Fellowship.

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

1 Introduction

Dynamically typed languages like Scheme [5] 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 known as *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 [9, 14] 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 type systems [4, 6] provide the benefits of static typing for dynamically typed languages. Like a static type checker, a soft type checker infers types for variables 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. These run-time checks indicate potential program errors, enabling programmers to detect errors prior to program execution. Soft type checking minimizes the number of run-time checks in the compiled code, enabling dynamically typed languages to attain the efficiency of statically typed languages like ML.

We have developed a practical soft type system for R4RS Scheme [5], a modern dialect of Lisp. *Soft Scheme* is based on an extension of the Hindley-Milner polymorphic type discipline that requires no programmer supplied type annotations. It presents types in a natural type language that is easy for non-expert 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 inserts only 10% of the run-time checks that are necessary for safe execution without soft typing. We have observed consequent speedups of up to 70% over a high quality Scheme compiler.

Soft Scheme is based on a soft type system designed by Cartwright and Fagan for an idealized functional language [4, 6]. Their type system extends Hindley-Milner typing with 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, exceptions, and top-level definitions. Finally, our system augments Scheme with pattern matching and type definition extensions that facilitate more precise type assignment.

[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 R4RS Scheme programs. When applied to a program, the type checker prints 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 the 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 expression (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:

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

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

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 program requires three run-time checks at primitive operations, one in each of the definitions of c, d, and e. The modified program shows the locations of the run-time checks:

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

An unnecessary CHECK-car is inserted because the type checker is unable to establish from the type of b that car is applied to a pair.[3] CHECK-add1 indicates that this primitive may fail when applied to some element of a, as indeed it will. Finally, ERROR-add1 indicates that this occurrence of add1 never succeeds—if it is ever reached, it will fail.

2.1 Presentation Types

Our type checker infers types in an encoded representation that reduces a limited form of subtyping to polymorphism. Soft Scheme decodes these types into more natural *presentation types* for programmers. Presentation types can precisely describe a rich collection of subsets of the data domain, yet they are simple enough for non-expert programmers to interpret. Conventional Hindley-Milner type systems form types from constants (*e.g. num*, *nil*), constructors (*e.g. cons*, *->*), and type variables (X). Our presentation types also include recursive types and a restricted form of union type.

Every presentation type is a union of a finite number of type constants and type constructions, collectively called *prime types*. A type may also include a single type variable:

$$
\begin{array}{lll}
(\textit{Type}) & T & ::= \; (+\; P_1 \ldots P_n) \;\mid\; (+\; P_1 \ldots P_n \; X) \\
(\textit{Prime Type}) & P & ::= \; \textit{num} \;\mid\; \textit{nil} \;\mid\; \ldots \;\mid\; (\textit{cons}\; T_1\; T_2) \;\mid\; (T_1 \ldots T_n \;\text{->}\; T) \;\mid\; N \\
(\textit{Place Holder}) & N & ::= \; (\textit{not num}) \;\mid\; (\textit{not nil}) \;\mid\; \ldots \;\mid\; (\textit{not cons}) \;\mid\; (\textit{not} \;\text{->})
\end{array}
$$

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.

Types must satisfy two restrictions. First, each tag may be used at most once within a union. This requirement precludes types like *(+ (cons true false) (cons false true))*; the less precise type *(cons (+ true false) (+ true false))* must be used instead. Second, the same set of tags must always precede a particular type variable. For example, when *(+ num nil (cons* T_1 T_2*) X1)* appears as the type of some program variable or expression,

[3]The primitive car extracts the first component of a pair. Its only valid input is a pair, but b's type *(list num) = (rec ([Y (+ nil (cons num Y))]) Y)* includes *nil*.

every other type in which *X1* appears must list the tags *num*, *nil*, *cons*, and no others. Special constructions called *place holders* may be used to meet this requirement in types like *((+ false X2) –> (+ (not false) X2))*. This requirement ensures that the range of a type variable excludes any types built from preceding tags. When a type variable is the sole element of a union, it ranges over all types. We then write X rather than *(+ X)*.

Recursive types are introduced by first-order recursion equations:

$$(\textit{Recursive Type}) \qquad R \ ::= \ (\textit{rec}\ ([X_1\ T_1] \ldots [X_n\ T_n])\ T) \ \mid \ T$$

The type *(rec ([Y1 (+ nil (cons X Y1))]) Y1)* is the type of proper lists containing elements of type X. This type occurs so frequently that we abbreviate it *(list X)*. By convention, we name recursive type variables 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 l of type *(list X1)*, and applies f to every element of l. It returns a list of the results. Function member takes a key x and a list l, and searches l for an occurrence of x. It returns the first sublist starting with element x if one exists; otherwise, it returns false. Procedure read parses an "s-expression" from an input device, returning 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 more detailed examples.

2.2 Performance

Soft Scheme inserts run-time type checks only at uses of primitive operations that cannot be proven safe. Figure 1 summarizes run-time checking for the Scheme versions of the Gabriel Common Lisp bench marks.[4] The percentages indicate how frequently our system inserts run-time checks compared to conventional compilers. 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 real speedup that Soft Scheme can achieve over a high quality Scheme compiler (Chez Scheme 4.1t on an unloaded SparcStation 1). We have computed speedup with reference to Chez Scheme's optimize-level 2 which performs maximum optimization while retaining safety. The *potential* speedup is the maximum obtainable speedup, achieved by ruthlessly discarding all run-time checks (optimize-level 3).[5] The

[4] Obtained from the Scheme Repository at nexus.yorku.ca.

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

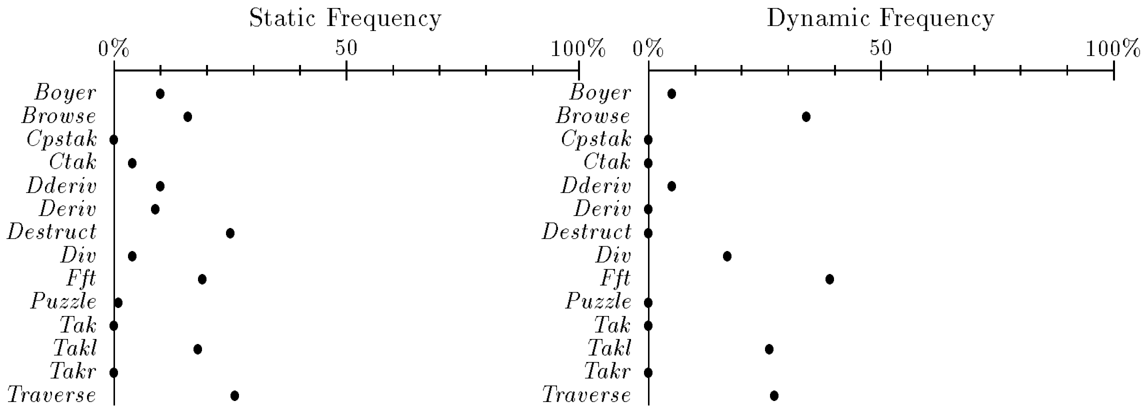

Figure 1: Reduction in Run-Time Checking for the Gabriel Bench Marks

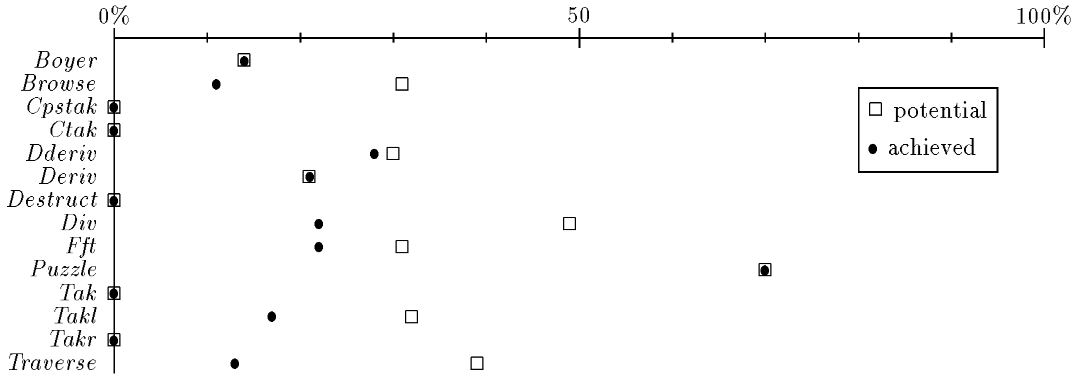

Figure 2: Speedup for the Gabriel Bench Marks

achieved speedup indicates the performance the soft typed program achieves without sacrificing safety. *Puzzle* illustrates the dramatic benefit (70% speedup) that can be obtained by removing run-time type checks from inner loops.

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. This reduction does not always lead to a significant performance improvement; it 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 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.

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 the following abstract syntax:

$$
\begin{array}{llcl}
(\mathit{Exp}) & e & ::= & v \mid (\mathbf{ap}\ e_1\ e_2) \mid (\mathbf{CHECK\text{-}ap}\ e_1\ e_2) \mid (\mathbf{let}\ ([x\ e_1])\ e_2) \\
(\mathit{Val}) & v & ::= & c \mid x \mid (\mathbf{lambda}\ (x)\ e)
\end{array}
$$

where $x \in \mathit{Id}$ are *identifiers* and $c \in \mathit{Const}$ are *constants*. *Const* includes both *basic constants* (numbers, #t, #f, '()) and *primitive operations* (add1, not, cons, car, etc). The keywords **ap** and **CHECK-ap** introduce *unchecked* and *checked* applications (explained 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 the answer error. For example, (**ap** CHECK-add1 #t) yields error. Similarly, **ap** and **CHECK-ap** introduce unchecked and checked *applications* that are undefined (*resp.* yield error) when their first subexpression is not a procedure. Appendix B contains a formal operational semantics for Core Scheme.

Designing a soft type system for Core Scheme is a challenging technical problem. Conventional Hindley-Milner type systems are too weak to infer precise types for dynamically typed languages because they presume that monotypes are disjoint. In contrast, values in dynamically typed programs belong to many different types, and dynamically typed programs routinely exploit this fact. The next two subsections define a collection of static types and a static type inference system that permits many union types to be expressed as terms in a free algebra, just like conventional Hindley-Milner types. *Flag variables* enable polymorphism to encode subtyping as subset on union types. The third subsection indicates how to translate these more general types into presentation types. The last subsection adapts this static type system to a soft type system for Core Scheme.

3.1 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:[6]

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

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. 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):

$$\kappa_1^{f_1} \vec{\sigma}_1 \cup \ldots \cup \kappa_n^{f_n} \vec{\sigma}_n \cup \Big(\alpha \mid \phi\Big)$$

[6] $\oplus$ denotes the coalesced tagged sum of two Scott domains, $\otimes$ denotes strict Cartesian product, $\to_{sc}$ is the strict continuous function space constructor, and $[\ ]_\perp$ is the lifting construction on domains.

where $\kappa \in \mathit{Tag} = \{\mathit{num}, \mathit{true}, \mathit{false}, \mathit{nil}, \mathit{cons}, \rightarrow\}$. Each prime type has a *flag* (f) 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. For example:[7]

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

To be well-formed, types must be *tidy*: each tag may be used at most once within a union. More precisely, a tidy type has the form:

$$\begin{array}{llcl}
(\mathit{Type}) & \sigma^X, \tau^X & ::= & \alpha^X \mid \emptyset^X \mid \mu\alpha^X.\,\tau^X \mid (\kappa^f \sigma_1^{\emptyset} \dots \sigma_n^{\emptyset})^X \cup \tau^{X \cup \{\kappa\}} \qquad (\kappa \notin X) \\
(\mathit{Flag}) & f & ::= & \varphi \mid \mathbf{-} \mid \mathbf{+}
\end{array}$$

$$\alpha, \beta \in \mathit{TypeVar} \qquad \varphi \in \mathit{FlagVar} \qquad X \in \mathbf{2}^{\mathit{Tag}}$$

where *TypeVar* and *FlagVar* are disjoint sets of *type variables* and *flag variables*. The arity n of each term $(\kappa^f \sigma_1^{\emptyset} \dots \sigma_n^{\emptyset})^X$ is determined by the arity of the tag κ. The superscript *labels* (X) attached to types are sets of tags that are *unavailable* for use in the type, and ensure that types are tidy. For example, the phrase:

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

is not a well-formed type because the term $(\mathit{num}^{\mathbf{-}})^{\{\mathit{num}\}}$ violates the restriction $\kappa \notin X$ in the formation of unions. Top-level types have label $\emptyset$. We usually omit labels when writing types. Similarly, an implementation of the type system need not manipulate labels.

Types represent regular trees with the tags κ and tidy union operator $\cup$ as internal nodes. Recursive types $\mu\alpha.\,\tau$ represent infinite regular trees. 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 $\mu\beta.\,\mathit{nil}^{\mathbf{+}} \cup (\mathit{cons}^{\mathbf{+}}\,\alpha\,\beta) \cup \emptyset$, denoting proper lists of α, is a common recursive type.

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[8] map from type variables to types and from flag variables to flags. St means the simultaneous replacement of every free variable in the type or flag t by its image under S. A type τ' is an *instance* of type scheme $\forall\vec{\alpha}\vec{\varphi}.\,\tau$ under substitution S:

$$(\prec) \qquad \tau' \prec_S \forall\vec{\alpha}\vec{\varphi}.\,\tau$$

if $\mathrm{Dom}(S) = \{\vec{\alpha}\vec{\varphi}\}$ and $S\tau = \tau'$. For example, $((\mathit{num}^{\mathbf{+}} \cup \emptyset) \rightarrow^{\mathbf{+}} (\mathit{num}^{\mathbf{+}} \cup \emptyset)) \cup \emptyset$ is an instance of $\forall\alpha.\,(\alpha \rightarrow^{\mathbf{+}} \alpha) \cup \emptyset$ under the substitution $\{\alpha \mapsto (\mathit{num}^{\mathbf{+}} \cup \emptyset)\}$.

[7] We use infix notation and write $(\sigma_1 \rightarrow^f \sigma_2) \cup \dots$ rather than $(\rightarrow^f \sigma_1 \sigma_2) \cup \dots$ for procedure types.

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

As Rémy [17, 18] discovered, polymorphism can encode a limited form of subtyping. In our framework, we use polymorphism to express supersets and subsets of types. The type scheme $\forall\alpha.\, num^{\boldsymbol{+}} \cup \alpha$ may be instantiated to any type that is a superset of $num^{\boldsymbol{+}} \cup \phi$:

$$\begin{array}{l} num^{\boldsymbol{+}} \cup \phi \\ num^{\boldsymbol{+}} \cup true^{\boldsymbol{+}} \cup \phi \\ num^{\boldsymbol{+}} \cup true^{\boldsymbol{+}} \cup false^{\boldsymbol{+}} \cup \phi \\ \qquad\quad \vdots \end{array}$$

The type scheme $\forall\varphi_1\varphi_2.\, num^{\varphi_1} \cup nil^{\varphi_2} \cup \phi$ may be instantiated to any type that denotes a subset of $num^{\boldsymbol{+}} \cup nil^{\boldsymbol{+}} \cup \phi$. There are four such types:

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

We use this ability to encode supersets and subsets in assigning types to basic constants and primitives. Basic constants and the outputs of primitives may have any type that is a superset of their natural type. Unchecked primitives require that their inputs be subsets of the largest allowable input types.

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 $\boldsymbol{-}$ and ϕ for invalid inputs. Positive unions use $\boldsymbol{+}$ for "present" outputs and variables for "absent" fields. For example:

$$\begin{array}{rcl} \mathit{TypeOf}(\mathsf{0}) & = & \forall\alpha.\, num^{\boldsymbol{+}} \cup \alpha \\ \mathit{TypeOf}(\mathsf{add1}) & = & \forall\alpha_1\alpha_2\varphi.\, ((num^{\varphi} \cup \phi) \rightarrow^{\boldsymbol{+}} (num^{\boldsymbol{+}} \cup \alpha_1)) \cup \alpha_2 \\ \mathit{TypeOf}(\mathsf{number?}) & = & \forall\alpha_1\alpha_2\alpha_3.\, (\alpha_1 \rightarrow^{\boldsymbol{+}} (true^{\boldsymbol{+}} \cup false^{\boldsymbol{+}} \cup \alpha_2)) \cup \alpha_3 \end{array}$$

Checked primitives have similar types to their unchecked counterparts, but never use $\boldsymbol{-}$ or ϕ since they accept all values. For example,

$$\mathit{TypeOf}(\mathsf{CHECK\text{-}add1}) \;=\; \forall\alpha_1\alpha_2\alpha_3\varphi.\, ((num^{\varphi} \cup \alpha_3) \rightarrow^{\boldsymbol{+}} (num^{\boldsymbol{+}} \cup \alpha_1)) \cup \alpha_2.$$

3.2 Static Type Inference

Figure 3 defines a *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 our 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 error, or (iv) gets *stuck* (reaches a non-value normal form, like ($\mathsf{ap\ add1\ \#t}$)). *Type soundness* ensures that typable programs yield answers of the expected type and do not get stuck.

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

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

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

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

Figure 3: Type Inference

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

Proof. We use Wright and Felleisen's technique based on subject reduction [24]. ∎

3.3 Translating to Presentation Types

The types assigned by this type system are awkward to read. To present more palatable types to the programmer, we define a translation into the succinct presentation type language introduced in Section 2.1. This translation eliminates (i) variables used to encode subtyping, and (ii) flags. Our decoding translation is a straightforward adaptation of that described by Cartwright and Fagan [4, 6].

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.[9] A soft type checker automates this process.

Figure 4 defines a *soft type inference* system for Core Scheme programs. This system both assigns types and computes a transformed expression in which some unchecked primitives and applications are replaced by checked ones. A *soft typing* $A \vdash_s e \Rightarrow e' : \tau$ states that in type environment A, the expression e transforms to e' of 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 variables called *absent*

[9]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.

$$(\textbf{const})\ \frac{\tau \prec_S \mathit{SoftTypeOf}(c)}{A \vdash_s c \Rightarrow (\mathit{empty}\{S\tilde{\nu} \mid \tilde{\nu} \in \mathrm{Dom}(S)\} \rightarrow c,\ \textsf{CHECK-}c) : \tau} \qquad (\textbf{var})\ \frac{\tau \prec_S A(x)}{A \vdash_s x \Rightarrow x : \tau}$$

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

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

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

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

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

Figure 4: Soft Type Inference

---

variables ($\tilde{\nu}$). Absent variables record uses of unchecked primitives that may not be safe. Wherever the function *TypeOf* places a $-$ flag or $\emptyset$ term 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,

$$\mathit{SoftTypeOf}(\textsf{add1}) \;=\; \forall \alpha_1 \alpha_2 \tilde{\alpha}_3 \varphi.\, ((\mathit{num}^{\varphi} \cup \tilde{\alpha}_3) \rightarrow^{+} (\mathit{num}^{+} \cup \alpha_1)) \cup \alpha_2.$$

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 $\mathit{true}^{+} \cup \emptyset$ is not empty, 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 function *empty* used by rules **const** and **ap** in Figure 4 determines whether every member of a set of types is empty. This is a simple syntactic test for the absence of $+$ flags.

To ensure that non-empty argument types propagate through intervening **let**-expressions to primitives that may require run-time checks, *absent variables are not generalized by SoftClose*. Hence, substitutions map absent flag variables to *absent flags* ($\tilde{f}$) and absent type variables to *absent types* ($\tilde{\tau}$):

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

A simple example demonstrates why this restriction is necessary. The expression:

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

requires add1 to be checked because it is applied to the value #t. In typing this expression, add1 is assigned type $((\mathit{num}^{\varphi'} \cup \tilde{\alpha}') \rightarrow^{+} \ldots)$ where $\tilde{\alpha}'$ and φ' are fresh variables. Suppose

SoftClose generalized absent variables. Generalizing the type of **add1** would yield type scheme $\forall\tilde{\alpha}''\varphi''.((num^{\varphi''} \cup \tilde{\alpha}'') \rightarrow^{+} \ldots)$ for inc. In typing the application (**ap** inc #t), $\tilde{\alpha}''$ would be instantiated as $true^{+} \cup \beta$. 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 that allows absent variables to be generalized.

Appendix C presents a Correspondence theorem to establish the correctness of this soft type system.

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. We also present two extensions to Scheme that enable more precise type inference, and 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)*. Variable arity procedure types use recursive argument lists. 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. **CHECK-lambda** is inserted where a **lambda**-expression requires argument count checking. In practice, we find that such argument count checks often 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 [23]. For Scheme, all expressions are values except those that contain an application of a non-primitive function 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. The first case is handled as usual by inserting CHECK-call/cc. 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. 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 into nested **letrec**-expressions and typed in the usual manner.

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 1]) (if (pair? x) (car x) 2))
```

no check should be necessary at car. However, as our type system assigns types to variables, 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:[10]

```
(let ([x 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 · · ·)*. 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. Five of the R4RS Scheme primitives 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; append when the last element is not a list; and make-vector without an initial value. All but the last can be handled by using an overly restrictive type and inserting unnecessary run-time checks. We prefer to ban uses of make-vector without an initial value. Overall, we feel that our type language provides a good balance between simplicity and expressiveness.

[10] The expression (**match** e $[pat_1\ e_1] \ldots [pat_n\ e_n]$) compares the value of e against *patterns* $pat_1 \ldots pat_n$. Any variables 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. Pattern (pat_1 . pat_2) matches a pair whose components match pat_1 and pat_2. Pattern _ matches anything.

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:

$$\frac{A \vdash e_1 : \tau_1 \qquad A \vdash e_2 : \tau_2 \qquad A \vdash e_3 : \tau_2}{A \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau_2}$$

Also applications require the types of arguments to match the types the function expects (see rules **ap** and **Cap** in Figure 3). 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 function f:

$$(\textbf{define}\ \mathsf{f}\ (\textbf{lambda}\ (\mathsf{x})\ (\textbf{if}\ P\ \mathsf{x}\ \#\mathsf{f})))$$

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. An argument to f is forced to include false^{+} in its type, even if that type would otherwise not include false^{+}. 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:

$$(\textbf{let}\ ([\mathsf{f}\ \mathsf{add1}])\ (\textbf{lambda}\ (\mathsf{x})\ (\mathsf{f}\ \mathsf{x})\ (*\ 2\ \mathsf{x})))$$

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 [15] 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. They yield only a small improvement in precision because existing extensions to recursive types [10, 22] do not yield *principal types*. This minor improvement comes at the cost of significantly less efficient type inference. We continue to investigate adaptations of structural subtyping.

Assignment: Because assignment interferes with polymorphism, and therefore with subtyping, assignment can be a major source of imprecision. Scheme includes both assignable variables, set by **set!**, and assignable pairs, set by set-car! and set-cdr!. Assignments to local variables seldom cause trouble. However, assignments to global variables 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 variable 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 [4, 6]. 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 [17]. Their system has essentially the same types as Section 3.1 describes, except that all type variables must have label $\emptyset$. 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:

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

Such types are difficult to decode into a natural type language that does not use flags. The 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 tidy unions, recursive types, subtype constraints, intersection types, and conditional types. While we have not yet had the opportunity to analyze their system in detail, it is clear that their system assigns more precise types to some programs than our system does. But both their timing results and the complexity of their algorithm indicate that it is much slower than ours. If their system can be extended to imperative languages (assignment and control) and implemented efficiently, it could serve as the 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 [7, 8, 16, 20, 21]. 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 a phrase prevents its use as a reliable indicator of potential program errors. 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 [3, 11, 12, 13, 19]. 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.

6 Future Work

We have developed the first practical soft type system for Scheme. Soft Scheme is available by anonymous FTP from titan.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 in file public/wright/match.tar.Z.

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 soft module systems to enable separate type checking and compilation of different parts of a program.

Acknowledgements

Kent Dybvig extended Chez Scheme overnight to permit mixing checked and unchecked primitives. Without his assistance, we could not have observed real execution time speedup.

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), to appear.

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

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

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

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

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] Tiuryn, J., and Wand, M. Type reconstruction with recursive types and atomic subtyping. In *TAPSOFT '93: Theory and Practice of Software Development* (Berlin, Heidelber, New York, apr 1993), M.-C. Gaudel and J.-P. Jouannaud, Eds., Springer, pp. 686–701.

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

[24] 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.

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))))))
;; ((Z1 -> Z2) (list Z1) -> (list Z2))

(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]
      [(? procedure?) (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

The following reduction relation specifies an operational semantics for Core Scheme (neglecting pairs, which are easy to add):

$$
\begin{array}{llcll}
(\mathit{let}) & E[(\textbf{let}\ ([x\ v])\ e)] & \longmapsto & E[e[x \mapsto v]] & \\
(\beta_{\mathrm{v}}) & E[(\mathit{apply}\ (\textbf{lambda}\ (x)\ e)\ v)] & \longmapsto & E[e[x \mapsto v]] & \\
(\delta) & E[(\mathit{apply}\ c\ v)] & \longmapsto & E[\delta(c, v)] & \text{if } c \in \mathit{Prim} \text{ and } \delta(c,v) \in \mathit{Val} \\
(\mathit{wrong\text{-}prim}) & E[(\mathit{apply}\ c\ v)] & \longmapsto & \mathsf{error} & \text{if } c \in \mathit{Prim} \text{ and } \delta(c,v) = \mathsf{error} \\
(\mathit{wrong\text{-}ap}) & E[(\textbf{CHECK-ap}\ c\ v)] & \longmapsto & \mathsf{error} & \text{if } c \notin \mathit{Prim}
\end{array}
$$

where *apply* is either **ap** or **CHECK-ap**, and $\mathit{Prim} \subset \mathit{Const}$ is the set of primitive operations. The reduction relation depends on a definition of *evaluation contexts*:

$$E \ ::= \ [\,] \ | \ (\mathit{apply}\ E\ e) \ | \ (\mathit{apply}\ v\ E) \ | \ (\textbf{let}\ ([x\ E])\ e)$$

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, as every non-value expression can be uniquely decomposed into an evaluation context and a redex.

The partial function $\delta : \mathit{Prim} \times \mathit{ClosedVal} \rightharpoonup (\mathit{ClosedVal} \cup \{\mathsf{error}\})$ interprets the application of primitives. *ClosedVal* is the set of closed values; error is not a value. For all checked primitives CHECK-c, we require that $\delta(\mathsf{CHECK}\text{-}c, v)$ be defined for all closed values v. For unchecked primitives, δ may be undefined at some arguments, and must never yield error. For corresponding pairs of unchecked and checked primitives (c, CHECK-c), we require that $\delta(c, v)$ and $\delta(\mathsf{CHECK}\text{-}c, v)$ agree for all v, except that $\delta(c, v)$ is undefined when $\delta(\mathsf{CHECK}\text{-}c, v)$ yields error:

$$\delta(\mathsf{CHECK}\text{-}c, v) = \begin{cases} \delta(c, v) & \text{if } \delta(c,v) \in \mathit{ClosedVal} \\ \mathsf{error} & \text{if } \delta(c,v) \text{ is undefined.} \end{cases}$$

When δ returns error for the application of a checked primitive, error immediately becomes the answer of the program via reduction *wrong-prim*. Similarly, checked applications of non-procedural values yield error via reduction *wrong-ap*.

With unchecked operations, evaluation can lead to a normal form that is neither a value nor error. 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*. Let $\longmapsto\!\!\!\!\rightarrow$ be the reflexive and transitive closure of $\longmapsto$. 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 error, or (iv) become stuck.

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

C Correspondence

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 error, 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[11] closure of the following relation:

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

Soft type checking lifts invalid programs that get stuck to error.

Theorem C.1 (Correspondence) *If $\emptyset \vdash_s e \Rightarrow e' : \tau$ then:*

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

Proof. We need two lemmas. The first shows that e' has type τ in the static type system.

Lemma C.2 *If $A \vdash_s e \Rightarrow e' : \tau$ then $A \vdash e' : \tau$.*

This lemma is proved by induction over the structure of the deduction $A \vdash_s e \Rightarrow e' : \tau$.

Second, a program that has fewer checked operations performs the same evaluation steps, but may become stuck sooner.

Lemma C.3 (Simulation) *For $e_1 \sqsubseteq e_1'$:*

1. $e_1 \longmapsto e_2 \Leftrightarrow e_1' \longmapsto e_2'$ *where* $e_2 \sqsubseteq e_2'$;

2a. $e_1 \longmapsto \mathsf{error} \Rightarrow e_1' \longmapsto \mathsf{error}$; *2b.* e_1 *is stuck* $\Rightarrow (e_1' \longmapsto \mathsf{error}$ *or* e_1' *is stuck*$)$;

3a. $e_1' \longmapsto \mathsf{error} \Rightarrow (e_1 \longmapsto \mathsf{error}$ *or* e_1 *is stuck*$)$; *3b.* e_1' *is stuck* $\Rightarrow e_1$ *is stuck.*

This lemma is proved by case analysis on the structure of the expressions.

With these lemmas, we can establish the theorem. From $\emptyset \vdash_s e \Rightarrow e' : \tau$ we have $e \sqsubseteq e'$ by induction and case analysis of the rules in Figure 4. By induction with simulation:

1a. $e \longmapsto\!\!\!\!\!\rightarrow v \Leftrightarrow e' \longmapsto\!\!\!\!\!\rightarrow v'$ *where* $v \sqsubseteq v'$; *1b.* e *diverges* $\Leftrightarrow e'$ *diverges*;

2a. $e \longmapsto\!\!\!\!\!\rightarrow \mathsf{error} \Rightarrow e' \longmapsto\!\!\!\!\!\rightarrow \mathsf{error}$; *2b.* e *gets stuck* $\Rightarrow (e' \longmapsto\!\!\!\!\!\rightarrow \mathsf{error}$ *or* e' *gets stuck*$)$;

3a. $e' \longmapsto\!\!\!\!\!\rightarrow \mathsf{error} \Rightarrow (e \longmapsto\!\!\!\!\!\rightarrow \mathsf{error}$ *or* e *gets stuck*$)$; *3b.* e' *gets stuck* $\Rightarrow e$ *gets stuck.*

But $\emptyset \vdash e' : \tau$ by Lemma C.2, hence by Type Soundness e' cannot get stuck. Simplifying *2b* and discarding *3b*, we obtain the theorem. ∎

[11] 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.