A Practical Soft Type System for Scheme

ANDREW K. WRIGHT
NEC Research Institute
and
ROBERT CARTWRIGHT
Rice University

A *soft type system* infers types for the procedures and data structures of dynamically typed programs. Like conventional static types, soft types express program invariants and thereby provide valuable information for program optimization and debugging. A soft type *checker* uses the types inferred by a soft type system to eliminate run-time checks that are provably unnecessary; any remaining run-time checks are flagged as potential program errors. *Soft Scheme* is a practical soft type checker for R4RS Scheme. Its underlying type system generalizes conventional Hindley-Milner type inference by incorporating recursive types and a limited form of union type. Soft Scheme accommodates all of R4RS Scheme including uncurried procedures of fixed and variable arity, assignment, and continuations.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Run-time checks, soft typing

1. INTRODUCTION

Dynamically typed languages such as Scheme [Clinger and Rees 1991] permit program operations to be defined over any computable subset of the data domain. To ensure safe execution, primitive operations confirm via run-time checks that their arguments belong to appropriate subsets called *types*. The simple argument types that primitive operations enforce (e.g., *num*, *bool*, *cons*) induce more complex types for the inputs of defined procedures (e.g., *list of num*, *bool or num or cons*). Scheme programmers typically have strong intuitive ideas about the types of their

---

This article expands on the *Lisp and Functional Programming* conference paper [Wright and Cartwright 1994] by including material drawn from the first author's Ph. D. thesis [Wright 1994].
Authors' addresses: A. K. Wright, NEC Research Institute, 4 Independence Way, Princeton, NJ 08831; R. Cartwright, Rice University, Houston, TX 77251-1892.
The first author was supported by NECI and, while at Rice, by a National Defense Science and Engineering Graduate Fellowship and Rice University. The second author was supported by NSF grant CCR-9122518 and Texas Advanced Technology Program grant 003604-014.

program's procedures and data structures, but dynamically typed languages traditionally offer no tools to discover, verify, or express such types.

Static type systems such as the Hindley-Milner [Hindley 1969; Milner 1978] type discipline can infer types for programs lacking explicit type annotations. A static type system consists of a set of syntactic rules ensuring that program operations are applied to the expected *form* of data.[1] Languages with a static type discipline support safe program execution without most of the run-time checks present in dynamically typed languages. But such a discipline limits the subsets of the data domain (i.e., types) over which program operations can be defined. Static typing is therefore unsuitable for dynamically typed languages such as Scheme.

A *soft* type system [Cartwright and Fagan 1991; Fagan 1990] infers types for the procedures and data structures of dynamically typed programs. Like conventional static types, soft types express program invariants and thereby provide valuable information for program optimization and debugging. A soft type *checker* uses the types inferred by a soft type system to eliminate run-time checks that are provably unnecessary; any remaining run-time checks are flagged as potential program errors.

We have developed a practical soft type system and soft type checker for R4RS Scheme [Clinger and Rees 1991], a modern dialect of Lisp. *Soft Scheme* is based on an extension of the Hindley-Milner static 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 precise to provide useful diagnostic assistance to programmers. For our benchmarks, the type checker typically eliminates 90% of the run-time checks that are necessary for safe execution without soft typing. Consequently, most soft typed programs run 10 to 15% faster than their dynamically typed counterparts, and selected examples run more than twice as fast.

The type system underlying Soft Scheme is a refinement and extension of a soft type system designed by Cartwright and Fagan [1991] and Fagan [1990] for an idealized functional language. 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, it uses a different representation for types that integrates polymorphism smoothly with union types and is more computationally efficient. This 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 system addresses the "grubby" features of a real programming language that Cartwright and Fagan's study ignored. In particular, it handles uncurried procedures of fixed and variable arity, assignment, and first-class continuations. Finally, our system augments Scheme with pattern matching and type definition extensions that facilitate more precise type assignment. We present a detailed comparison to Cartwright and Fagan's system in Section 6.1.1.

---

[1]It is important to separate static type *systems* which are sound deductive systems from the *heuristic* type checking rules in popular programming languages such as C, C++, and Pascal. Heuristic checking does not prevent program operations from being applied to the wrong form of data.

1.1 An Illustration

Soft Scheme assigns types to arbitrary Scheme programs by inserting explicit run-time checks at applications of *primitive* operations. Given an input program, the type checker writes a typed version of the program with explicit run-time checks to an output file and displays only a summary of the inserted run-time type checks. The programmer can interactively inspect the types assigned to program expressions.

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

```
(define flatten
  (λ (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. 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 yielded by the expression '(1 (2) 3), which abbreviates (cons 1 (cons (cons 2 '()) (cons 3 '()))). The primitive operation cons constructs pairs, which 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. Flatten returns a proper list of type (*list* (+ (*not cons*) (*not nil*) *X1*)). The elements of the result list have type *X1* and do not include pairs or the empty list.

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

Now suppose we add the following lines to our program:

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

Soft type checking the extended program yields the following summary:

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

The extended program requires three run-time checks at primitive operations, one in each of the definitions of c, d, and e. The typed program produced as 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 run-time check is inserted at car because b's type

$$(\mathit{list}\ \mathit{num}) = (\mathit{rec}\ ([Y\ (+\ \mathit{nil}\ (\mathit{cons}\ \mathit{num}\ Y))])\ Y)$$

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.

1.2 Outline

The next section presents a formal development of a soft type system for a simple language based on the λ-calculus. Since the *internal* types inferred by this soft type system are expressed in an unfamiliar type language that is difficult for programmers to interpret directly, Section 3 presents a simpler language of *presentation* types and translations between *internal* and *presentation* types. The examples above use presentation types. Section 4 extends the type system to address various language constructs that are found in realistic programming languages such as Scheme. In Section 5 we discuss experiences and performance results obtained from our prototype soft type system for Scheme. Finally, Section 6 places our work in the context of other work on soft typing, optimization, and static type systems.

2. A SOFT TYPE SYSTEM FOR CORE SCHEME

As the first step in a formal description of a soft type system, we define an idealized, dynamically typed, call-by-value language called *Core Scheme* that embodies the essence of Scheme.

2.1 Syntax and Semantics

Core Scheme has expressions ($e \in \mathit{Exp}$) and values ($v \in \mathit{Val}$) of the forms

$$
\begin{array}{rcl}
e & ::= & v \mid (\mathbf{ap}\ e_1\ e_2) \mid (\mathbf{CHECK\text{-}ap}\ e_1\ e_2) \\
 & & \mid (\mathbf{if}\ e_1\ e_2\ e_3) \mid (\mathbf{let}\ ([x\ e_1])\ e_2)
\end{array} \qquad (\mathit{Exp})
$$

$$
v \ ::= \ c \mid x \mid (\boldsymbol{\lambda}\ (x)\ e) \qquad (\mathit{Val})
$$

where $x \in \mathit{Id}$ are *identifiers*, and $c \in \mathit{Const}$ are *constants*. *Const* includes both *basic constants* (numbers, #t, #f, '() $\in \mathit{Basic} \subset \mathit{Const}$) and *primitive operations* ($p \in \mathit{Prim} \subset \mathit{Const}$). Primitive operations include both *unchecked primitives* such as add1, car, and cons, as well as *checked primitives* such as CHECK-add1, CHECK-car, and CHECK-cons. The keywords **ap** and **CHECK-ap** introduce *unchecked* and *checked* applications, which we explain below. The free identifiers $FV(e)$ and bound identifiers of an expression are defined as usual, with $\boldsymbol{\lambda}$- and **let**-expressions binding their identifiers. The **let**-expression binds x in e_2 but not e_1, i.e., **let**-bindings are not recursive. Following Barendregt [1984], we adopt the convention that bound identifiers are always distinct from free identifiers in distinct expressions, and we identify expressions that differ only by a consistent renaming of the bound identifiers. *Programs* are closed expressions.

To incorporate run-time checks, Core Scheme includes both *unchecked* as well as *checked* versions of every primitive operation. Invalid applications of unchecked primitives, such as (**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 error messages. For example, (**ap** CHECK-add1 #t) yields an error message. Similarly, **ap** and **CHECK-ap** introduce unchecked and checked *applications* that are undefined (respectively, yield an error message) when their first subexpression is not a procedure. For example, the expression (**ap** 1 2) is meaningless, while (**CHECK-ap** 1 2) terminates execution with an error message like "Error: 1 is not a procedure."

We use reduction semantics [Felleisen and Hieb 1992] to specify the operational behavior of Core Scheme programs. Figure 1 defines the single-step reduction relation $\longmapsto$ for Core Scheme (neglecting pairs, which are easy to add). The reduction relation $\longmapsto$ 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,[3] as every expression that is not a value can be uniquely decomposed into an evaluation context and a redex.

Rules β_{v} and *check*-β_{v} reduce ordinary and checked applications of $\boldsymbol{\lambda}$-expressions by substitution. The notation $e[x \mapsto v]$ denotes the expression formed by substituting v for free x in e, renaming bound variables of v as necessary to avoid capture. Rule *let* reduces **let**-expressions by substitution. Rules if_1 and if_2 reduce **if**-expressions according to whether the test value is the special constant #f. Rules

[3] Our theorems also hold for a language that does not specify the evaluation order, such as Scheme.

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

$$
\begin{array}{rcl}
E & ::= & [\,] \bigm| (\mathbf{ap}\ E\ e) \bigm| (\mathbf{ap}\ v\ E) \bigm| (\mathbf{CHECK\text{-}ap}\ E\ e) \bigm| (\mathbf{CHECK\text{-}ap}\ v\ E) \\
 & & \bigm| (\mathbf{if}\ E\ e_1\ e_2) \bigm| (\mathbf{let}\ ([x\ E])\ e)
\end{array}
$$

Fig. 1. Reduction semantics for Core Scheme.

δ_1, δ_2, *check*-δ_1, and *check*-δ_2 use the partial function

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

to interpret the application of primitives. $\mathit{ClosedVal} \subset \mathit{Val}$ is the set of closed values, and check is an error message returned by primitive operations that fail. For unchecked primitives, δ may be undefined at some arguments. For all unchecked primitives p, we require that a checked primitive $\mathsf{CHECK}\text{-}p$ exist if $\delta(p, v)$ is not defined for every $v \in \mathit{ClosedVal}$. A checked primitive behaves the same way as its unchecked counterpart, except it returns check when the unchecked primitive is undefined:

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

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

The reduction relation $\longmapsto$ is the basis of program evaluation. Programs evaluate according to the relation $\longmapsto\!\!\!\!\rightarrow$, which is the reflexive and transitive closure of $\longmapsto$. Answers are values or the special token check, which is returned by programs that apply checked operations to invalid arguments.

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 a expression is *stuck*:

$$\mathit{Stuck} \;=\; \left\{ \begin{array}{ll} E[(\mathbf{ap}\ p\ v)] & \text{where } \delta(p,v) \text{ is undefined,} \\ E[(\mathbf{CHECK\text{-}ap}\ p\ v)] & \text{where } \delta(p,v) \text{ is undefined,} \\ E[(\mathbf{ap}\ c\ v)] & \text{where } c \notin \mathit{Prim} \end{array} \right\}.$$

We say that e *diverges* when there is an infinite reduction sequence $e \longmapsto e' \longmapsto e'' \longmapsto \ldots$. The following lemma asserts that all closed expressions either (1) yield an answer that is a closed value, (2) diverge, (3) yield check, or (4) become stuck.

Lemma 2.1.1. *For all closed expressions e, either $e \longmapsto\!\!\!\!\rightarrow v$ where v is closed, e diverges, $e \longmapsto\!\!\!\!\rightarrow \mathsf{check}$, or $e \longmapsto\!\!\!\!\rightarrow e'$ where e' is stuck.*

Proof. The proof is a routine induction on the length of the reduction sequence, using case analysis on the structure of the expression e. For a proof of a similar theorem, see Felleisen [1991]. □

Type-safe implementations of dynamically typed languages like Core Scheme interpret all occurrences of primitive operations in source programs 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.

2.2 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 semantic types, and dynamically typed programs routinely exploit this fact. To accommodate these overlapping types, a soft type system for Core Scheme should include union types and use the following rule to infer types for applications:

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

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 directly on union types and attempt to find an alternative method of inferring types. Aiken et al. [1994] have pursued this approach, but both its computational complexity and its practical behavior are significantly worse than Hindley-Milner typing. We elected instead to modify Hindley-Milner typing to accommodate union types and subtyping without compromising its practical efficiency.[4]

To combine union types and subtyping with Hindley-Milner polymorphism, we adapt an encoding for record subtyping pioneered by Wand [1987; 1991] and refined by Rémy [1989; 1991]. Our encoding permits many union types to be expressed as terms in a free algebra, as with conventional Hindley-Milner types. *Flag variables*

[4]Ordinary Hindley-Milner typing relies on simple unification of finite terms, which can be implemented in linear time [Martelli and Montanari 1976; Paterson and Wegman 1978]. Our modification of Hindley-Milner typing requires unification of infinite terms, for which no linear algorithm is known. But practical implementations of simple unification use nonlinear algorithms that are faster for small types than the linear algorithms. Practical implementations of infinite unification are faster still because they omit the "occurs check."

enable polymorphism to encode subtyping as subset on union types. Types are inferred by a simple variant of the standard Hindley-Milner algorithm. When displaying types to the programmer, we decode the inferred types into more natural union types. Our type system thereby provides the *illusion* of a polymorphic union type system based on ordinary union types. The illusion is imperfect: occasionally the decoded types do not match what informal reasoning about natural union types 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 system based on a variation of Rémy's encoding. Following that, we adapt this static type system to a soft type system for Core Scheme.

2.3 Static Types

To construct a static type system for Core Scheme, we partition the data domain into disjoint subsets for which the primitive operations are (mostly) closed. Informally, the primitive operations of Core Scheme induce the following partitioning of the domain:

$$\begin{aligned}\mathbf{D} \;=\; & \mathit{numbers} \\ \cup\; & \{\#\mathsf{t}\} \\ \cup\; & \{\#\mathsf{f}\} \\ \cup\; & \{\text{'}()\} \\ \cup\; & \{\langle v_1, v_2\rangle \mid v_1 \in \mathbf{D}_1 \subseteq \mathbf{D} \text{ and } v_2 \in \mathbf{D}_2 \subseteq \mathbf{D}\} \\ \cup\; & \{f \mid f(v_1) \in \mathbf{D}_2 \subseteq \mathbf{D} \text{ for all } v_1 \in \mathbf{D}_1 \subseteq \mathbf{D}\}.\end{aligned}$$

For Core Scheme, all numbers inhabit the same partition because primitive operations such as division and exponentiation can produce small integer, big integer, rational, real, or complex answers. (In a soft type system for an ML-like language where there are distinct operations for integers and reals, we would use distinct partitions *int* and *real*.) The constants #t, #f, and '() each inhabit their own partitions. The partitions containing pairs and procedures are further subdivided as the components of each $(\mathbf{D}_1, \mathbf{D}_2)$ are partitioned in the same way. Appendix B includes a precise definition of the data domain as a reflexive domain equation. The domain equation reflects the above partitioning of the data domain.

The static types for Core Scheme reflect the partitioning of the data domain. We define the set of static types in two stages. We first define a set of *pretypes*, of which the static types for Core Scheme are a subset. Informally, every pretype (σ, τ) is a disjoint union of zero or more partitions ($\kappa^f \sigma_1 \ldots \sigma_n$, also written $\kappa^f \vec{\sigma}$) followed by either a single type variable (α) or the empty type ($\emptyset$):

$$\sigma, \tau \quad ::= \quad \kappa_1^{f_1} \vec{\sigma} \cup \ldots \cup \kappa_n^{f_n} \vec{\tau} \cup \Big(\alpha \;\Big|\; \emptyset\Big)$$

where

$$f \quad ::= \quad \mathbf{+} \;\Big|\; \mathbf{-} \;\Big|\; \varphi$$

denotes a *flag*; $\varphi \in \mathit{FlagVar}$ is a *flag variable*; and κ is a *tag*. Types represent regular trees with tags constructing internal nodes; the parentheses and union symbol $\cup$ are merely syntax to enhance readability. In a partition $\kappa^f \sigma_1 \ldots \sigma_n$, the constructor κ has arity $n+2$: flag f, types $\sigma_1 \ldots \sigma_n$, and type τ are its arguments. Tags, denoted by κ, designate partitions of the data domain. The tags *num*, *true*, *false*, and *nil* identify the partitions containing numbers, #t, #f, and '(), respectively. The tags *cons* and $\rightarrow$ identify the partitions containing pairs and procedures. Each partition has a flag f (written above the tag) that indicates whether the partition is part of the union type. A flag **+** indicates the partition is present; **–** indicates that it is absent; and a flag variable (φ) indicates the partition may be present or absent depending on how the flag variable is instantiated. In general, a static type with free flag variables designates a finite set of possible types corresponding to the possible instantiations of the free flag variables. The following are some examples of types:

$num^{+} \cup \emptyset$	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.*

We use infix notation and write $(\sigma_1 \rightarrow^f \sigma_2)$ rather than $(\rightarrow^f \sigma_1 \sigma_2)$ for procedure partitions.

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 universe as explained below. Tidiness permits us to find and represent the pairwise unification between two sets of types (i.e., two union types) by performing a single unification step. The following class of grammars defines the (tidy) static types (σ^X, τ^X) of Core Scheme:

$$\sigma^{\emptyset}, \tau^{\emptyset} \ ::= \ \alpha^{\emptyset} \mid \emptyset^{\emptyset} \mid (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^{\emptyset} \cup \tau^{\{\kappa\}} \mid \mu\alpha^{\emptyset}.\tau^{\emptyset}$$

$$\sigma^{X}, \tau^{X} \ ::= \ \alpha^{X} \mid \emptyset^{X} \mid (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^{X} \cup \tau^{X \cup \{\kappa\}} \qquad (\kappa \notin X)$$

where $X \in \mathbf{2}^{Tag}$ is a *label*. Labels enforce tidiness by specifying sets of tags that are *not* available for use in the type. For example, the phrase

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

is not a tidy type because the term $(num^{-})^{\{num\}}$ violates the restriction $\kappa \notin X$ in the formation of types. Labels also limit the universe for type variables. In the Hindley-Milner type system, all type variables may range over the entire universe of types. In our system, the range of a type variable that appears in a union type excludes types built from the tags of partitions preceding it, i.e., the tags in its label. That is, in a type

$$(\kappa_1^{f_1} \vec{\sigma}_1)^{\emptyset} \cup \ldots \cup (\kappa_n^{f_n} \vec{\sigma}_n)^{\{\kappa_1, \ldots, \kappa_{n-1}\}} \cup \alpha^{\{\kappa_1, \ldots, \kappa_n\}},$$

the universe for type variable α excludes partitions constructed from $\kappa_1 \ldots \kappa_n$. For instance, in the type

$$num^{+} \cup true^{-} \cup (cons^{+} \sigma_1 \sigma_2) \cup \alpha,$$

the range of type variable α excludes numbers, #t, and all pairs. The types assigned to program identifiers and expressions have label $\emptyset$. We usually omit labels when writing types, as they can be easily reconstructed. Similarly, an implementation of type inference need not manipulate labels.

Recursive types $\mu\alpha.\,\tau$ represent infinite regular trees [Amadio and Cardelli 1990]. 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 such as $\mu\alpha.\,\alpha$ are not types. The type $\mu\alpha.\, nil^{+} \cup (cons^{+}\,\tau\;\alpha) \cup \phi$, which denotes proper lists.

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 (1) types that differ only in the order of union components and (2) types that denote different representations of the empty type:

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

It is easy to verify that this quotient preserves tidiness.

To accommodate polymorphism and subtyping, we introduce *type schemes*:

$$\Sigma \;::=\; \forall\vec{\nu}.\,\tau$$

where $\nu \in (\mathit{TypeVar} \cup \mathit{FlagVar})$ denotes a type or flag variable. The type scheme $\forall\vec{\nu}.\,\tau$ binds type and flag variables $\{\vec{\nu}\}$ 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 map from type variables to types and from flag variables to flags. $S\tau$ (respectively, Sf) means the simultaneous replacement of every free variable in type τ (respectively, flag f) by its image under S. Since types are required to be tidy, the application $S\tau$ makes sense only when S preserves the labeling of τ. A type τ' is an *instance* of type scheme $\forall\vec{\nu}.\,\tau$ under substitution S, written

$$\tau' \prec_S \forall\vec{\nu}.\,\tau,$$

if $\mathrm{Dom}(S) = \{\vec{\nu}\}$ and $S\tau = \tau'$. For example, the type

$$((num^{+} \cup \phi) \to^{+} (num^{+} \cup \phi)) \cup \phi$$

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

In our framework, we use polymorphism both to express conventional polymorphic types and to express subsets of tidy union types. For instance, 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 $num^{+} \cup nil^{+} \cup \phi$. There are four such types:

$$\begin{aligned} &num^{+} \cup nil^{+} \cup \phi \\ &num^{-} \cup nil^{+} \cup \phi \\ &num^{+} \cup nil^{-} \cup \phi \\ &num^{-} \cup nil^{-} \cup \phi. \end{aligned}$$

By using polymorphic flag variables for the inputs of primitive operations and procedures, we can simulate subtyping at applications while still using unification

0	:	$\forall\alpha.\, num^{+} \cup \alpha$
#t	:	$\forall\alpha.\, true^{+} \cup \alpha$
add1	:	$\forall\alpha_1\alpha_2\varphi.\, ((num^{\varphi} \cup \emptyset) \rightarrow^{+} (num^{+} \cup \alpha_1)) \cup \alpha_2$
number?	:	$\forall\alpha_1\alpha_2\alpha_3.\, (\alpha_1 \rightarrow^{+} (true^{+} \cup false^{+} \cup \alpha_2)) \cup \alpha_3$
not	:	$\forall\alpha_1\alpha_2\alpha_3.\, (\alpha_1 \rightarrow^{+} (true^{+} \cup false^{+} \cup \alpha_2)) \cup \alpha_3$
cons	:	$\forall\alpha_1\alpha_2\alpha_3\alpha_4.\, (\alpha_1 \rightarrow^{+} ((\alpha_2 \rightarrow^{+} (cons^{+}\, \alpha_1\, \alpha_2)) \cup \alpha_3)) \cup \alpha_4$
car	:	$\forall\alpha_1\alpha_2\alpha_3\varphi.\, (((cons^{\varphi}\, \alpha_1\, \alpha_2) \cup \emptyset) \rightarrow^{+} \alpha_1) \cup \alpha_3$

Fig. 2. Static types for unchecked constants.

to equate the procedure's input type and the argument's type. A procedure with type scheme $\forall\varphi_1\varphi_2.\, (num^{\varphi_1} \cup nil^{\varphi_2} \cup \emptyset) \rightarrow^{+} \tau$ 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 $\forall\alpha.\, num^{+} \cup \alpha$. This may be instantiated to any type that is a superset of $num^{+} \cup \emptyset$:

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

This ensures that expressions such as (**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 differs 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 (for reasons explained presently) and $-$ and $\emptyset$ for invalid inputs. Positive unions use $+$ for "present" outputs and variables for "absent" fields. Figure 2 presents the types for some of the constants and unchecked primitive operations of Core Scheme.

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, primitives CHECK-add1 and CHECK-car have the following type schemes:

CHECK-add1	:	$\forall\alpha_1\alpha_2\alpha_3\varphi.\, ((num^{\varphi} \cup \alpha_3) \rightarrow^{+} (num^{+} \cup \alpha_1)) \cup \alpha_2$
CHECK-car	:	$\forall\alpha_1\alpha_2\alpha_3\alpha_4\varphi.\, (((cons^{\varphi}\, \alpha_1\, \alpha_2) \cup \alpha_4) \rightarrow^{+} \alpha_1) \cup \alpha_3.$

$$\frac{\tau \prec_S \mathit{TypeOf}(c)}{A \vdash c : \tau} \qquad (\mathbf{const}_\vdash)$$

$$\frac{\tau \prec_S A(x)}{A \vdash x : \tau} \qquad (\mathbf{id}_\vdash)$$

$$\frac{A \vdash e_1 : (\tau_2 \to^f \tau_1) \cup \emptyset \qquad A \vdash e_2 : \tau_2}{A \vdash (\mathbf{ap}\ e_1\ e_2) : \tau_1} \qquad (\mathbf{ap}_\vdash)$$

$$\frac{A \vdash e_1 : (\tau_2 \to^f \tau_1) \cup \tau_3 \qquad A \vdash e_2 : \tau_2}{A \vdash (\mathbf{CHECK\text{-}ap}\ e_1\ e_2) : \tau_1} \qquad (\mathbf{CHECK\text{-}ap}_\vdash)$$

$$\frac{A[x \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\lambda\ (x)\ e) : (\tau_1 \to^{+} \tau_2) \cup \tau_3} \qquad (\mathbf{lam}_\vdash)$$

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

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

$$\mathit{Close}(\tau, A) = \forall \vec{\nu}.\, \tau \text{ where } \{\vec{\nu}\} \subseteq FV(\tau) - FV(A)$$

Fig. 3. Static type inference rules.

These type schemes are obtained by replacing $\emptyset$ in the type schemes of the unchecked primitives with quantified type variables.

As discussed above, polymorphic flag variables in the inputs to procedures provide subtyping at applications of those procedures. But for a procedure such as add1 that has type scheme

$$\forall \alpha_1 \alpha_2 \varphi.\, ((\mathit{num}^{\varphi} \cup \emptyset) \to^{+} (\mathit{num}^{+} \cup \alpha_1)) \cup \alpha_2, \tag{1}$$

the only subtypes of its input type are $\mathit{num}^{+} \cup \emptyset$ and the trivial type $\mathit{num}^{-} \cup \emptyset = \emptyset$. Since add1 accepts only one kind of input, an alternative type scheme is

$$\forall \alpha_1 \alpha_2.\, ((\mathit{num}^{+} \cup \emptyset) \to^{+} (\mathit{num}^{+} \cup \alpha_1)) \cup \alpha_2. \tag{2}$$

This type scheme does not have the trivial type $\emptyset$ as a subtype of its input type. Both (1) and (2) are valid type schemes for add1. We use the first type scheme for add1 because the second causes *reverse flow*, a phenomenon discussed in Section 5.5.1.

2.4 Static Type Checking

Figure 3 defines a *static type* system that assigns types to Core Scheme expressions. Type environments (A) are finite maps from identifiers to type schemes. $A[x \mapsto \Sigma]$ denotes the functional extension or update of A at x to Σ. $FV(\Sigma)$ returns the free type and flag variables of a type Σ. FV extends pointwise to type environments. The *typing* $A \vdash e : \tau$ states that expression e has type τ in type environment A. When A is the empty map, we write simply $\vdash e : \tau$.

In rules $\mathbf{ap}_\vdash$ and $\mathbf{CHECK\text{-}ap}_\vdash$, the first antecedent allows the type of e_1 to include an arbitrary flag f on the constructor $\to$. As with the type schemes assigned to primitives, this typing avoids the problem of reverse flow (Section 5.5.1). Alternative typing rules for applications could use $+$ rather than f, but fewer expressions would be typable.

A static type system ensures type safety by assigning types only to programs that cannot lead to meaningless operations (i.e., cannot get stuck). A static type system that meets this criterion is *sound*. To ensure soundness, *Const*, δ, and *TypeOf* must satisfy the following *typability* conditions. For every c, τ, τ', f, and v,

(1) when $c \in \mathit{Prim}$, if $(\tau' \rightarrow^f \tau) \cup \emptyset \prec_S \mathit{TypeOf}(c)$ and $\vdash v : \tau'$, then either $\delta(c, v) = \mathsf{check}$ or $\delta(c, v) = v'$ and $\vdash v' : \tau$;

(2) when $c \notin \mathit{Prim}$, there exists no S, τ', τ such that $(\tau' \rightarrow^f \tau) \cup \emptyset \prec_S \mathit{TypeOf}(c)$.

Condition (1) requires that δ be defined for all primitive operations of functional type and arguments of matching type, and restricts the set of results that δ may produce. Condition (2) ensures that only primitive operations, and not basic constants, may have functional type.

Given the typability conditions, we can establish soundness. Recall that Section 2.1 defines a reduction relation $\longmapsto\!\!\!\!\rightarrow$ for which every program either (1) yields an answer v, (2) diverges, (3) yields the error message check, or (4) gets *stuck* (Lemma 2.1.1). *Type soundness* ensures that typable programs yield answers of the expected type and do not get stuck.

THEOREM 2.4.1 (TYPE SOUNDNESS). *If $\vdash e : \tau$, then either $e \longmapsto\!\!\!\!\rightarrow v$ and $\vdash v : \tau$, or e diverges, or $e \longmapsto\!\!\!\!\rightarrow \mathsf{check}$.*

PROOF. We use Wright and Felleisen's [1994] technique based on subject reduction. The proof relies on two main lemmas. The first lemma, *Subject Reduction*, states that evaluation preserves typing. The second lemma states that stuck programs are not typable.

LEMMA 2.4.2 (SUBJECT REDUCTION). *If $\vdash e_a : \sigma$ and $e_a \longmapsto e_b$ then $\vdash e_b : \sigma$.*

LEMMA 2.4.3 (UNTYPABILITY OF STUCK PROGRAMS). *If $e \in \mathit{Stuck}$ then there is no τ such that $\vdash e : \tau$.*

Given these lemmas, we can prove Type Soundness. From *Uniform Evaluation* (Lemma 2.1.1), either $e \longmapsto\!\!\!\!\rightarrow v$ where v is closed, e diverges, $e \longmapsto\!\!\!\!\rightarrow \mathsf{check}$, or $e \longmapsto\!\!\!\!\rightarrow e'$ where $e' \in \mathit{Stuck}$. Since $\vdash e : \tau$, if $e \longmapsto\!\!\!\!\rightarrow v$ then $\vdash v : \tau$ by *Subject Reduction*. All that remains to show is that e cannot get stuck. Suppose it does; that is, suppose that $e \longmapsto\!\!\!\!\rightarrow e'$ where $e' \in \mathit{Stuck}$. Then $\vdash e' : \tau$ by *Subject Reduction*. But this contradicts *Untypability of Stuck Programs*; hence e cannot get stuck. $\square$

To establish *Subject Reduction*, we use some obvious facts about deductions:

(1) if $A \vdash C[e] : \tau$ then there exist A', τ' such that $A' \vdash e : \tau'$ (C is a context);

(2) if there are no A', τ' such that $A' \vdash e : \tau'$, then there are no A, τ such that $A \vdash C[e] : \tau$.

These follow from the facts that (1) there is exactly one inference rule for each expression form and (2) each inference rule requires a proof for each subexpression of the expression in its conclusion.

A key lemma that we use in the proof of *Subject Reduction* is a *Replacement* lemma, adapted from Hindley and Seldin [1986, p. 181]. This allows the replacement of one of the subexpressions of a typable expression with another subexpression of the same type, without disturbing the type of the overall expression.

Lemma 2.4.4 (Replacement). *If*

(1) $\mathcal{D}$ *is a deduction concluding* $A \vdash C[e_1] : \tau$,
(2) $\mathcal{D}_1$ *is a subdeduction of* $\mathcal{D}$ *concluding* $A' \vdash e_1 : \tau'$,
(3) $\mathcal{D}_1$ *occurs in* $\mathcal{D}$ *in the position corresponding to the hole in* C*, and*
(4) $A' \vdash e_2 : \tau'$,

then $A \vdash C[e_2] : \tau$.

A *Substitution* lemma is the key to showing *Subject Reduction* for reductions involving substitution.

Lemma 2.4.5 (Substitution). *If* $A[x \mapsto \forall\vec{\alpha}\vec{\varphi}.\tau] \vdash e : \tau'$ *and* $x \notin \mathrm{Dom}(A)$ *and* $A \vdash v : \tau$ *and* $\{\vec{\alpha}\vec{\varphi}\} \cap FV(A) = \emptyset$ *then* $A \vdash e[x \mapsto v] : \tau'$.

The proof of *Substitution* proceeds by induction on the length of the proof of $A[x \mapsto \forall\vec{\alpha}\vec{\varphi}.\tau] \vdash e : \tau'$ and by case analysis on the last step. The proof is an adaptation of our similar lemma for an ordinary Hindley-Milner type system [Wright and Felleisen 1994, Lemma 4.4].

Proof of Subject Reduction. The proof proceeds by case analysis according to the reductions of Figure 1. Note that reductions δ_2 and *check*-δ_2 do not yield expressions and hence need not be considered.

Case $E[(\mathbf{ap}\ p\ v)] \longmapsto E[\delta(p,v)]$ *where* $\delta(p,v) \in \mathit{Val}$. Since there exist A and τ such that $A \vdash (\mathbf{ap}\ p\ v) : \tau$, we have $A \vdash p : (\tau' \to^f \tau) \cup \phi$ and $A \vdash v : \tau'$ by $\mathbf{ap}_\vdash$. By the typability conditions for constants, we have $A \vdash \delta(c,v) : \tau$, and hence we obtain $\vdash E[\delta(c,v)] : \sigma$ by *Replacement*.

Case $E[(\mathbf{CHECK\text{-}ap}\ p\ v)] \longmapsto E[\delta(p,v)]$ *where* $\delta(p,v) \in \mathit{Val}$. This is similar to the previous case.

Case $E[(\mathbf{if}\ v\ e_1\ e_2)] \longmapsto E[e_1]$ *where* $v \neq \#\mathsf{f}$. Since A and τ exist such that $A \vdash (\mathbf{if}\ v\ e_1\ e_2) : \tau$, we have $A \vdash e_1 : \tau$ by $\mathbf{if}_\vdash$. Hence $\vdash E[e_1] : \sigma$ by *Replacement*.

Case $E[(\mathbf{if}\ \#\mathsf{f}\ e_1\ e_2)] \longmapsto E[e_2]$. Since there exist A and τ such that $A \vdash (\mathbf{if}\ \#\mathsf{f}\ e_1\ e_2) : \tau$, we have $A \vdash e_2 : \tau$ by $\mathbf{if}_\vdash$. Hence $\vdash E[e_2] : \sigma$ by *Replacement*.

Case $E[(\mathbf{ap}\ (\boldsymbol{\lambda}\ (x)\ e)\ v)] \longmapsto E[e[x \mapsto v]]$. Since there exist A and τ such that $A \vdash (\mathbf{ap}\ (\boldsymbol{\lambda}\ (x)\ e)\ v) : \tau$, we have $A \vdash v : \tau'$ and $A \vdash (\boldsymbol{\lambda}\ (x)\ e) : (\tau' \to^f \tau) \cup \phi$ by $\mathbf{ap}_\vdash$. From the latter, we have $A[x \mapsto \tau'] \vdash e : \tau$ (and $f = \mathbf{+}$) by $\mathbf{lam}_\vdash$. Hence we obtain $A \vdash e[x \mapsto v] : \tau$ by *Substitution*, and $\vdash E[e[x \mapsto v]] : \sigma$ by *Replacement*.

Case $E[(\mathbf{CHECK\text{-}ap}\ (\boldsymbol{\lambda}\ (x)\ e)\ v)] \longmapsto E[e[x \mapsto v]]$. This is similar to the previous case.

Case $E[(\mathbf{let}\ ([x\ v])\ e)] \longmapsto E[e[x \mapsto v]]$. Since there exist A and τ such that $A \vdash (\mathbf{let}\ ([x\ v])\ e) : \tau$, we have $A \vdash v : \tau'$ and $A[x \mapsto \mathit{Close}(\tau', A)] \vdash e : \tau$ by $\mathbf{let}_\vdash$. As $\mathit{Close}(\tau', A) = \forall\vec{\alpha}\vec{\varphi}.\,\tau'$ where $\{\vec{\alpha}\vec{\varphi}\} = FV(\tau') - FV(A)$, we have $A \vdash e[x \mapsto v] : \tau$ by *Substitution*. Hence we have $\vdash E[e[x \mapsto v]] : \sigma$ by *Replacement*.

This completes the proof of *Subject Reduction*. □

Proof of Untypability of Stuck Programs. The proof proceeds by case analysis on the form of the stuck program.

Case $E[(\mathbf{ap}\ p\ v)]$ *where* $\delta(p,v)$ *is Undefined.* It suffices to show that there are no A, τ such that $A \vdash (\mathbf{ap}\ p\ v) : \tau$. Suppose otherwise. Then by $\mathbf{ap}_{\vdash}$, $A \vdash p : (\tau' \rightarrow^{f} \tau) \cup \emptyset$ and $A \vdash v : \tau'$ for some τ', f. But then by the typability conditions, $\delta(p,v)$ is defined, which contradicts the premise of this case.

Case $E[(\mathbf{CHECK\text{-}ap}\ p\, v)]$ *where* $\delta(p,v)$ *is Undefined.* This is similar to the previous case.

Case $E[(\mathbf{ap}\ c\ v)]$ *where* $c \notin Prim$. It suffices to show that no A, τ exist such that $A \vdash (\mathbf{ap}\ c\ v) : \tau$. Suppose otherwise. Then by $\mathbf{ap}_{\vdash}$, $A \vdash c : (\tau' \rightarrow^{f} \tau) \cup \emptyset$ and $A \vdash v : \tau'$ for some τ', f, contradicting the typability conditions for basic constants.

This completes the proof of *Untypability of Stuck Programs*. □

2.5 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 type-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.[5] A soft type checker automates this process.

Figure 4 defines a *soft type* 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* $A \mathrel{|\!\!\!\Rightarrow} e \Rightarrow e' : \tau$ states that in type environment A, expression e transforms to e' such that 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* ($\tilde{\nu} \in AbsVar = AbsTypeVar \cup AbsFlagVar$). 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 $\tilde{\varphi} \in AbsFlagVar$ or absent type variable $\tilde{\alpha} \in AbsTypeVar$. For example, *SoftTypeOf*(`add1`) is

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

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

$$\begin{array}{lcl} \tilde{f} & \in & \{f \mid FV(f) \subset AbsFlagVar\} \\ \tilde{\tau} & \in & \{\tau \mid FV(\tau) \subset AbsVar\}. \end{array}$$

[5] The same process cannot be used with statically typed languages such as ML because the Hindley-Milner type discipline does not provide implicit union types. One must also introduce 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.

$$\frac{\tau \prec_S \mathit{SoftTypeOf}(c) \qquad \mathit{Empty}\{S\tilde{\nu} \mid \tilde{\nu} \in \mathrm{Dom}(S)\}}{A \models c \Rightarrow c : \tau} \qquad (\mathbf{OKconst}_{\models})$$

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

$$\frac{\tau \prec_S A(x)}{A \models x \Rightarrow x : \tau} \qquad (\mathbf{id}_{\models})$$

$$\frac{A \models e_1 \Rightarrow e_1' : (\tau_2 \rightarrow^f \tau_1) \cup \tilde{\tau}_3 \qquad A \models e_2 \Rightarrow e_2' : \tau_2 \qquad \mathit{Empty}\{\tilde{\tau}_3\}}{A \models (\mathbf{ap}\ e_1\ e_2) \Rightarrow (\mathbf{ap}\ e_1'\ e_2') : \tau_1} \qquad (\mathbf{OKap}_{\models})$$

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

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

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

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

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

$$\mathit{SoftClose}(\tau, A) = \forall\vec{\nu}.\tau \text{ where } \{\vec{\nu}\} \subseteq FV(\tau) - (FV(A) \cup \mathit{AbsTypeVar} \cup \mathit{AbsFlagVar})$$

Fig. 4. Soft type inference rules.

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 nonempty type in the type assignment process, then the primitive operation whose type introduced that type variable 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 the type $\mathit{true}^{+} \cup \emptyset$ contains the element #t, 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 $\mathbf{OKconst}_{\models}$ and $\mathbf{OKap}_{\models}$ in Figure 4 determines whether every member of a set of types and flags can be instantiated to the empty type or the absent flag $-$. For a set of types and flags s, $\mathit{Empty}(s)$ is defined as

$$\mathit{Empty}(s) \;=\; \begin{cases} \mathit{false} & \text{if } \exists\, f \in s \text{ such that } f = +; \\ \mathit{false} & \text{if } \exists\, \tau \in s \text{ such that } \tau = \kappa^{+}\vec{\sigma} \cup \tau'; \\ \mathit{false} & \text{if } \exists\, \tau \in s \text{ such that } \tau = \kappa^{f}\vec{\sigma} \cup \tau' \text{ and } \neg\mathit{Empty}(\tau'); \\ \mathit{true} & \text{otherwise.} \end{cases}$$

Rules $\mathbf{OKconst}_{\models}$ and $\mathbf{OKap}_{\models}$ require that *Empty* hold for that part of the unchecked primitive's input type corresponding to undefined inputs. Rules $\mathbf{const}_{\models}$ and $\mathbf{ap}_{\models}$ have no such restriction—it is always possible to insert a run-time check. A type inference algorithm that inserts a minimal number of run-time checks chooses rules $\mathbf{OKconst}_{\models}$ and $\mathbf{OKap}_{\models}$ over rules $\mathbf{const}_{\models}$ and $\mathbf{ap}_{\models}$ whenever possible.

As a **let**-bound procedure may be used in several different ways, each use may necessitate different run-time checks in the procedure body. For instance, in the expression[6]

```
(let ([inc (λ (x) (ap add1 x))])
  (begin
    (ap inc 0)
    (ap inc #t)))
```

the use of inc in (**ap** inc 0) necessitates no run-time checks. The second use of inc requires a run-time check at add1. Hence a **let**-bound procedure must include the union of all run-time checks required by its uses. To ensure this, *absent variables are not generalized by SoftClose*. The above example demonstrates why this restriction is necessary. In typing the **let**-bound expression, add1 is assigned type $((num^{\varphi'} \cup \tilde{\alpha}') \rightarrow^{+} \ldots)$ where $\tilde{\alpha}'$ and φ' are fresh variables. Suppose *SoftClose* naively 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 2.6 describes a better method of inserting checks that performs some extra bookkeeping so that absent variables may be generalized.

To establish the correctness of our soft type system, we must show that all programs can be soft typed and that inserting run-time checks does not change the behavior of programs other than to cause programs that would get stuck to yield check instead. The following theorem establishes that all programs can be soft typed.

Theorem 2.5.1 (Applicability). *For all programs e, there exist e', τ such that $\models e \Rightarrow e' : \tau$.*

Proof. The proof proceeds by induction over the structure of typing derivations, using a strengthened induction hypothesis to accommodate terms with free identifiers. A key fact used in the proof is that it is always possible to find a unifying substitution for two types that contain no uses of $-$ or $\emptyset$. Since none of the soft type schemes for primitives nor the soft typing rules use these constants, it is always possible to pick types for primitives such that a type derivation for the program can be constructed. For a full proof, see the first author's thesis [Wright 1994]. □

To relate the behavior of source programs to the behavior of transformed programs, let $e \sqsubseteq e'$ mean that e' may have checked operation(s) where e has unchecked operation(s), but e and e' are otherwise the same. Specifically, $\sqsubseteq$ is the reflexive, transitive, and compatible[7] closure of the following relation:

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

[6]The expression (**begin** e_1 e_2) abbreviates (**ap** (λ (x) e_2) e_1) where $x \notin FV(e_2)$.
[7]The *compatible* closure of a relation R is $\{(C[e_1], C[e_2]) \mid (e_1, e_2) \in R$ for all contexts $C\}$. A context C is an expression with a hole in place of one subexpression.

The following theorem establishes that inserting run-time checks does not change the behavior of programs, other than to cause programs that would get stuck to yield check instead.

THEOREM 2.5.2 (CORRESPONDENCE). *For all* e, e', τ *such that* $\models e \Rightarrow e' : \tau$

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

PROOF. Recall that evaluation has four possible outcomes (Lemma 2.1.1). A program may (1) yield an answer v, (2) diverge, (3) yield check, or (4) get stuck. We first show that a program that has fewer checked operations performs the same evaluation steps, but may become stuck sooner.

LEMMA 2.5.3 (SIMULATION). *For* $e_1 \sqsubseteq e_1'$

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

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

For the forward direction of the theorem, from $\models e \Rightarrow e' : \tau$ we have $e \sqsubseteq e'$ by induction and case analysis of the soft typing rules in Figure 4. By induction with the first part of Simulation we have

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

All that remains is to show that e' cannot get stuck.

To show that e' cannot get stuck, we show that e' is typable in the static type system of Section 2.3. Static *Type Soundness* ensures that e' cannot get stuck. Let $\tilde{S}$ be the substitution with domain *AbsVar* that takes all absent type variables to $\emptyset$ and all absent flag variables to $-$.

LEMMA 2.5.4 (STATIC TYPABILITY). *If* $A \models e \Rightarrow e' : \tau$ *then* $\tilde{S}A \vdash e' : \tilde{S}\tau$.

This lemma is proved by induction over the structure of the deduction $A \models e \Rightarrow e' : \tau$. The proof exploits the close correlation between the type schemes assigned by *TypeOf* and *SoftTypeOf* and between the static typing rules and the soft typing rules. See Wright [1994] for a full proof of this lemma.

For the reverse direction of the theorem, again $e \sqsubseteq e'$. By induction with the second part of Simulation we have

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

This completes the proof of *Correspondence*. □

```
(let ([inc (λ (z) (ap add1 z))])
  (ap (λ (x)
        (begin
          (ap inc x)
          (ap sub1 x)
          (ap inc #t)))
      1))
```

Fig. 5. Program generating spurious run-time checks.

Correspondence also ensures that soft typed programs do not get stuck.

Corollary 2.5.5 (Safety). *For all e, e', τ such that $\models e \Rightarrow e' : \tau$, program e' does not get stuck.*

2.6 More Precise Type Assignment

In this section, we present two extensions to our soft type system that enable more precise type assignment and fewer run-time checks. The first extension permits generalization of absent flag and type variables. The second extension provides more subtyping by eliminating certain unnecessary absent variables and $+$-flags.

2.6.1 *Generalizing Absent Variables.* Some uses of a **let**-bound procedure may require that procedure to contain run-time checks, while other uses do not. For example, in the expression

```
(let ([inc (λ (z) (ap add1 z))])
  (ap inc 1)
  (ap inc #t))
```

the application of inc to #t requires a run-time check at add1 within inc. The application of inc to 1 by itself necessitates no run-time check. Cloning separate versions of a **let**-bound procedure for each use would allow run-time checks to be inserted only for uses of a procedure that require them. But uninhibited cloning is impractical, as it can exponentially increase program size.

Instead, we insert run-time checks in a **let**-bound procedure if any use of the procedure requires a check. The need for a run-time check is indicated by instantiation of an absent variable in the procedure's type scheme to a nonempty type. To collect run-time checks from different uses of a **let**-bound identifier, the soft type system described above prevents generalization of absent variables at a **let**-expression and thereby folds together the types of absent variables from different uses. Folding types from different uses together can yield less precise types and cause spurious run-time checks to be inserted. To illustrate how this occurs, consider the program in Figure 5. This program defines a procedure inc, correctly applies inc to x, correctly applies sub1 to x, and incorrectly applies inc to #t. In the absence of applications, inc has type scheme

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

Note that the absent variable $\tilde{\alpha}_3$ is free in this type scheme. Without the application (**ap** inc #t), this program requires no run-time checks as $\tilde{\alpha}_3$ is not instantiated further. But with the application (**ap** inc #t) as above, two run-time checks are required: one at add1 and one at sub1. The run-time check at add1 is required because the application (**ap** inc #t) instantiates $\tilde{\alpha}_3$ to $true^{\boldsymbol{+}} \cup \tilde{\alpha}_4$. Procedure inc now has type scheme

$$\forall \alpha_1 \alpha_2 \varphi. ((num^{\varphi} \cup true^{\boldsymbol{+}} \cup \tilde{\alpha}_4) \rightarrow^{\boldsymbol{+}} (num^{\boldsymbol{+}} \cup \alpha_1)) \cup \alpha_2$$

Hence the application (**ap** inc x) forces x to have type $num^{\boldsymbol{+}} \cup true^{\boldsymbol{+}} \cup \tilde{\alpha}_4$, and a spurious run-time check is inserted at sub1.

To avoid this kind of spurious run-time check, we extend the soft type system of the previous section to permit safe generalization of absent variables. We generalize absent variables as usual but record the set of types to which they are instantiated. A run-time check is required whenever any instance of a generalized absent variable is nonempty. Formally, we parameterize the soft type system over a set of substitutions Ψ. In the following discussion, we assume that the bound type variables of all type schemes in the deduction tree are distinct so that we can refer to these type variables in Ψ. To permit generalization of absent variables, we replace rules $\mathbf{let}_{\models\!\!\!>}$ and $\mathbf{id}_{\models\!\!\!>}$ with the following rules:

$$\frac{A \models\!\!\!> e_1 \Rightarrow e_1' : \tau_1 \qquad A[x \mapsto \mathit{Close}(\tau_1, A)] \models\!\!\!> e_2 \Rightarrow e_2' : \tau_2}{A \models\!\!\!> (\mathbf{let}\ ([x\ e_1])\ e_2) \Rightarrow (\mathbf{let}\ ([x\ e_1'])\ e_2') : \tau_2} \qquad (\mathbf{let}_2)$$

$$\frac{\tau \prec_S A(x) \qquad S|_{\mathit{AbsVar}} \in \Psi}{A \models\!\!\!> x \Rightarrow x : \tau} \qquad (\mathbf{var}_2)$$

The new $\mathbf{let}_2$ rule uses *Close* rather than *SoftClose* to generalize variables, thereby closing over absent variables. The new $\mathbf{var}_2$ rule records the instances of absent variables in Ψ. $S|_{\mathit{AbsVar}}$ means S restricted to absent variables. Finally, we add an additional clause to the definition of *Empty* as follows:

$$\mathit{Empty}(s) \;=\; \begin{cases} \mathit{false} & \text{if } \exists\, f \in s \text{ such that } f = \boldsymbol{+}; \\ \mathit{false} & \text{if } \exists\, \tau \in s \text{ such that } \tau = \kappa^{\boldsymbol{+}}\vec{\sigma} \cup \tau'; \\ \mathit{false} & \text{if } \exists\, \tau \in s \text{ such that } \tau = \kappa^{f}\vec{\sigma} \cup \tau' \text{ and } \neg\mathit{Empty}(\tau'); \\ \mathit{false} & \underline{\text{if } \exists\, \tilde{\nu} \in s \text{ such that } \neg\mathit{Empty}(S\tilde{\nu}) \text{ for some } S \in \Psi;} \\ \mathit{true} & \text{otherwise.} \end{cases}$$

An absent variable is empty if all of its (transitive) instances according to Ψ are empty.

To illustrate the new system, consider the program in Figure 5 again *with* the application (**ap** inc #t). Procedure inc now has type scheme

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

because the absent variable $\tilde{\alpha}_3$ is generalized by $\mathbf{let}_2$. The substitution set Ψ must contain two substitutions, one for each use of inc:

$$\Psi = \{\{\tilde{\alpha}_3 \mapsto \tilde{\alpha}_5\}, \{\tilde{\alpha}_3 \mapsto true^{\boldsymbol{+}} \cup \tilde{\alpha}_4\}\}.$$

Now x has type $num^{\boldsymbol{+}} \cup \tilde{\alpha}_5$. A run-time check is still required at add1, but no spurious run-time check is inserted at sub1.

2.6.2 *Type Swapping.* Recall that in the types assigned to primitives, the tails of unions in positive positions (which correspond to procedure outputs) are ordinary type variables. Flags in negative positions (which correspond to procedure inputs) are variables. Our soft type system relies on the generalization of these type and flag variables to support subtyping. But the types inferred for user-defined procedures do not always satisfy these properties. When a procedure's input type has a component with flag $\mathbf{+}$ rather than a flag variable, that component must be present in the types of all arguments passed to the procedure. Similarly, when a procedure's output union type ends in an absent type variable rather than an ordinary type variable, the procedure's output cannot be used as a wider type without forcing run-time checks to be inserted.

For example, in the following recursive definition[8]

```
(letrec ([f (λ (x) (ap f (ap sub1 x)))])
  f)
```

procedure f has soft type scheme

$$\forall \alpha_2 . ((num^{\mathbf{+}} \cup \tilde{\alpha}_1) \rightarrow^{\mathbf{+}} \alpha_2) \cup \tilde{\alpha}_3 . \tag{3}$$

(The absent variables $\tilde{\alpha}_1$ and $\tilde{\alpha}_3$ may be generalized if the extension from the previous section is used, but our illustration extends to this case.) Type variable $\tilde{\alpha}_3$ is an absent variable because the application (**ap** f ...) requires f to be a procedure. Note that the $\mathbf{+}$-flag on *num* appears in a negative (input) position, and the absent type variable $\tilde{\alpha}_3$ appears in a positive (output) position. Suppose f is used in a context that mixes it with a nonprocedural value:

```
(letrec ([f (λ (x) (ap f (ap sub1 x)))])        (4)
  (if e f #t))
```

Typing the **if**-expression requires instantiating $\tilde{\alpha}_3$ to $true^{\mathbf{+}} \cup \tilde{\alpha}_4$. The application (**ap** f ...) now receives an unnecessary run-time check because f's modified type scheme

$$\forall \alpha_2 . ((num^{\mathbf{+}} \cup \tilde{\alpha}_1) \rightarrow^{\mathbf{+}} \alpha_2) \cup true^{\mathbf{+}} \cup \tilde{\alpha}_4$$

now includes $true^{\mathbf{+}}$.

In our type system, there are often several syntactic types that denote the same set of values. For instance, the type

$$\forall \varphi \alpha_2 \alpha_3 . ((num^{\varphi} \cup \tilde{\alpha}_1) \rightarrow^{\mathbf{+}} \alpha_2) \cup \alpha_3 \tag{5}$$

denotes the same set of values as type (3). To see this, convert both to their equivalent static types:

$$\forall \alpha_2 . ((num^{\mathbf{+}} \cup \emptyset) \rightarrow^{\mathbf{+}} \alpha_2) \cup \emptyset \qquad \text{(static equivalent of 3)}$$
$$\forall \varphi \alpha_2 \alpha_3 . ((num^{\varphi} \cup \emptyset) \rightarrow^{\mathbf{+}} \alpha_2) \cup \alpha_3 \qquad \text{(static equivalent of 5)}$$

In the denotational semantics for types in Appendix B, these two types denote the same set of values. Hence it is safe to replace type (3) with type (5) in assigning a type to f. Unlike type (3), this new type has no $\mathbf{+}$-flags in negative positions, nor absent type variables in positive positions. By replacing the type for f in example (4) with type (5) when generalizing f's type, example (4) will no longer require an unnecessary run-time check.

[8] We use the standard method of typing **letrec**-expressions where the bound variables are polymorphic only in the body of the **letrec**-expression and not in the bindings [Milner et al. 1990].

In general, we can permit more precise type assignment by adding a *type-swapping* rule $\mathbf{swap}_{\vdash}$ to our type system:

$$\frac{A \vdash e : \tau_1 \qquad \mathcal{T}[\![\tau_1]\!] = \mathcal{T}[\![\tau_2]\!]}{A \vdash e : \tau_2} \qquad (\mathbf{swap}_{\vdash})$$

The function $\mathcal{T}[\![\cdot]\!]$ yields the set of values that its argument type denotes, according to the denotational semantics in Appendix B. The antecedent $\mathcal{T}[\![\tau_1]\!] = \mathcal{T}[\![\tau_2]\!]$ is a semantic condition that requires τ_1 and τ_2 to denote the same set of values in all type environments providing bindings for their free variables. We implement an approximation to this rule by performing the following replacements when generalizing types at **let**-expressions.

(1) Replace absent type variables that occur only positively in the type being generalized with fresh ordinary type variables.

(2) Replace **+**-flags that occur only negatively in the type being generalized with fresh ordinary flag variables.

As the above example indicates, this extension is particularly important for improving the types assigned to recursive procedures. Section 5.1 presents an example that illustrates the effect of this extension for a real program.

2.7 Inserting Errors

The soft type system described so far inserts run-time checks at primitive operations that *might* lead to errors. We can also use the information the system infers to determine whether primitive operations are ever applied to *valid* arguments. If the type for an occurrence of a primitive operation indicates that it may be applied to an invalid argument but is never applied to a valid argument, this primitive operation *will fail* if it is ever reached. This is a strong indication that the program may contain a bug. We flag such primitive operations by inserting special run-time checks called *errors*. An error is a run-time check that fails whenever it is applied.

To extend the semantics to include *errors*, for each primitive c we define an error version ERROR-c and require that $\delta(\mathsf{ERROR}\text{-}c, v) = \mathsf{check}$ for all v. We also add a new expression form (**ERROR-ap** e_1 e_2) for procedure applications that are errors. We extend the semantics appropriately so that **ERROR-ap** applications always terminate execution with answer check.

To determine whether a primitive operation is applied to a valid argument, we track instantiations of all variables, rather than just absent variables as in Section 2.6.1, by including all instantiating substitutions in Ψ:

$$\frac{\tau \prec_S A(x) \qquad S \in \Psi}{A \,\vDash\!\!\!>\, x \Rightarrow x : \tau} \qquad (\mathbf{var}_3)$$

We extend *Empty* to work for ordinary type and flag variables:

$$\mathit{Empty}(\nu) \;=\; \mathit{Empty}\{S\nu \mid S \in \Psi \text{ and } \nu \in \mathrm{Dom}(S)\}.$$

Now *Empty* can be used to determine whether that part of the data domain corresponding to a variable in a primitive operation's type is populated.

As before, a primitive such as add1 that has type scheme

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

requires a run-time check if $\tilde{\alpha}_3$ is not empty. But now we insert an *error* rather than an ordinary run-time check if the valid input domain of the primitive operation is empty. The valid input domain of add1 is num^{φ}. Hence we insert ERROR-add1 rather than CHECK-add1 if flag variable φ is empty. For example, in the program

```
(let ([inc (λ (z) (ap add1 z))])
  (ap inc 1)
  (ap inc #t))
```

primitive add1 receives an ordinary run-time check because Ψ includes $\{\varphi \mapsto +\}$ from the application (inc 1). In the program

```
(let ([inc (λ (z) (ap add1 z))])
  (ap inc #t))
```

φ is empty, and add1 receives an *error*:

```
(let ([inc (λ (z) (ap ERROR-add1 z))])
  (ap inc #t))
```

Inserting **ERROR-ap** for applications of nonprocedures is similar.

2.8 Implementing Type Inference

As our type system extends the Hindley-Milner system with union types and recursive types, we simply adapt conventional type inference algorithms for Hindley-Milner type inference to our system. Rémy [1992] describes the basic algorithms for unification, generalization, and instantiation of types in detail.

3. PRESENTING TYPES TO PROGRAMMERS

The soft type system developed in the previous section infers relatively precise types for Core Scheme programs. But these types are difficult for programmers to interpret due to the notational complexity introduced by (1) flags and (2) type variables used to encode subtyping. To present more intelligible types to the programmer, we define a translation to a set of *presentation types*. This translation eliminates flags and type variables used to encode subtyping.

3.1 Presentation Types

Presentation types include recursive types and tidy union types, just as internal types do. The *presentation types* (T) for Core Scheme are

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

where X denotes a type variable, and *union types* (U), *basic types* (B), and *place holders* (N) are

$$\begin{aligned} U \ &::= \ B \ \big| \ X \ \big| \ (+\ B_1 \ldots B_n\ [\, N_1 \ldots N_m\ X \,]\ ) \\ B \ &::= \ num \ \big| \ true \ \big| \ false \ \big| \ nil \ \big| \ (cons\ U_1\ U_2) \ \big| \ (U_1 \ {->}\ U_0) \\ N \ &::= \ (not\ num) \ \big| \ (not\ true) \ \big| \ (not\ false) \ \big| \ (not\ nil) \ \big| \ (not\ cons) \ \big| \ (not\ {->}). \end{aligned}$$

Unlike internal types, presentation types introduce recursive types using a set of first-order recurrence equations. The type (*rec* ($[X_1\ U_1] \ldots [X_n\ U_n]$) U_0) binds $X_1 \ldots X_n$ in $U_1 \ldots U_n$ and U_0 and denotes U_0 where

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

The order of the bindings is irrelevant, and we identify (*rec* () U) with U. The recursive 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*) (i.e., *list* is a type macro). By convention, we use names starting with Y for type variables bound by recursive types.

Basic types correspond to the partitions of the data domain (Section 2.3). A union type may consist of a single basic type, a single type variable, or several basic types. We identify union types that differ only in the order in which their components appear. We identify a union type (+ B) consisting of only a single basic type with that basic type B. Similarly, we identify a union type (+ X) consisting of only a type variable with that type variable X.

As with internal types, presentation types must be tidy. Each of the tags *num*, *true*, *false*, *nil*, *cons*, and –> may be used at most once within a union type (+ $B_1 \ldots B_n$). When a union type includes a type variable, the type variable's universe implicitly excludes any types constructed from the tags of $B_1 \ldots B_n$. Place holders serve to further constrain the universe of a type variable. If a union type includes place holders preceding a type variable, as in the type

(+ $B_1 \ldots B_n\ N_1 \ldots N_m\ X$),

then the universe for the type variable X also excludes any types constructed from the tags of $N_1 \ldots N_m$. For example, the type

(+ *true false X1*)

includes #t and #f. The universe of *X1* excludes the types *true* and *false*. The type

(+ (*not true*) (*not false*) *X2*)

does not include #t or #f. The universe of *X2* is the same as that of *X1*.

Type variables in a type may be free or quantified. Rather than use $\forall$ to indicate that a type variable is quantified, type variables that begin with an uppercase letter are considered quantified at the type in which they appear. Type variables that begin with lowercase letters indicate free type variables.

Following are some simple Scheme procedures to illustrate presentation types. The following procedure

```
(define f
  (λ (x)
    (if (null? x) '() (+ 1 x))))
```

returns the empty list (which has type *nil*) if passed the empty list, and a number if passed a number. It has type ((+ *num nil*) –> (+ *num nil*)). Another procedure with the same type is g:

```
(define g
  (λ (x)
    (if (null? x) 1 (begin (+ 1 x) '()))))
```

It returns a number if passed the empty list, and the empty list if passed a number.

Procedure types may include a type variable shared between the input and result. For example, procedures f2 and g2

```
(define f2                  (define g2
  (λ (x)                      (λ (x)
    (if (null? x)               (if (null? x)
        '()                         1
        (if (number? x)             (if (number? x)
            (+ 1 x)                     '()
            x))))                       x))))
```

both have type

$$((+\ num\ nil\ X1) \to (+\ num\ nil\ X1)). \tag{6}$$

Such shared type variables indicate a modicum of dependence of the result on the input. This type is interpreted as follows:

(1) Given a number, a procedure of type (6) may return a number or the empty list.

(2) Given an empty list, a procedure of type (6) may return a number or the empty list.

(3) Given an input of type *X1*, which excludes *num* and *nil*, a procedure of type (6) may return a number, the empty list, or a value of type *X1*.

In the first two cases, a result of type *X1* cannot be returned because the result must belong to (+ *num nil X1*) for every type *X1*. In particular, the result must belong to (+ *num nil*) when *X1* = (+), where (+) denotes the empty type that has no values (corresponding to the internal type $\emptyset$).

The following procedure from the introduction illustrates a presentation type that involves a union type, a recursive type, place holders, and a shared type variable:

```
(define flatten
  (λ (l)
    (cond [(null? l) '()]
          [(pair? l) (append (flatten (car l)) (flatten (cdr l)))]
          [else (list l)])))
```

This procedure has type

(*rec* ([*Y1* (+ *nil* (*cons Y1 Y1*) *X1*)])
  (*Y1* –> (*list* (+ (*not nil*) (*not cons*) *X1*)))).

This type indicates that flatten takes as input any tree and returns a list of the nonempty leaves of the tree. The leaves are any values in the input other than pairs or the empty list.

Following are the types for a few well-known Scheme functions. Several of these types use extended procedure types $(U_1 \ldots U_n \text{->} U_0)$ of arity n.

map	:	$((\mathit{X1} \text{->} \mathit{X2})\ (\mathit{list}\ \mathit{X1}) \text{->} (\mathit{list}\ \mathit{X2}))$
member	:	$(\mathit{X1}\ (\mathit{list}\ \mathit{X1}) \text{->} (+\ \mathit{false}\ (\mathit{cons}\ \mathit{X1}\ (\mathit{list}\ \mathit{X1}))))$
read	:	$(\mathit{rec}\ ([\mathit{Y1}\ (+\ \mathit{num}\ \mathit{nil} \ldots (\mathit{cons}\ \mathit{Y1}\ \mathit{Y1}))])$ $(\text{->} (+\ \mathit{eof}\ \mathit{num}\ \mathit{nil} \ldots (\mathit{cons}\ \mathit{Y1}\ \mathit{Y1}))))$
lastpair	:	$(\mathit{rec}\ ([\mathit{Y1}\ (+\ (\mathit{cons}\ \mathit{X1}\ \mathit{Y1})\ \mathit{X2})])$ $((\mathit{cons}\ \mathit{X1}\ \mathit{Y1}) \text{->} (\mathit{cons}\ \mathit{X1}\ (+\ (\mathit{not}\ \mathit{cons})\ \mathit{X2}))))$

The higher-order function map takes a function f of type $(\mathit{X1} \text{->} \mathit{X2})$ and a list x of type $(\mathit{list}\ \mathit{X1})$ 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 spine of pairs.

Appendix A contains additional examples of presentation types.

3.2 Displaying Presentation Types

To translate inferred internal types into presentation types, we must eliminate (1) flags and (2) type variables used to encode subtyping.

We define certain type and flag variables as *useless* with respect to a soft typing deduction for a complete program. In defining useless variables, we assume that all bound variables in the typing differ and are distinct from free variables. With respect to soft typing $\models e \Rightarrow e' : \tau$, 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 $\models e \Rightarrow e' : \tau$), 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 basic types. Components with flag **–** translate to place holders $(\mathit{not}\ \kappa)$ or are dropped entirely if the union ends in $\emptyset$.

To illustrate the translation, consider the following internal type scheme for flatten

$$\begin{aligned}\forall\alpha\varphi_1\varphi_2\varphi_3\varphi_4.\,(\mathit{rec}\ (&[y_1\ \ \mathit{nil}^{\varphi_1} \cup (\mathit{cons}^{\varphi_2}\, y_1\, y_1) \cup \alpha]\\ &[y_2\ \ \mathit{nil}^{+} \cup (\mathit{cons}^{+}(\mathit{nil}^{\varphi_3} \cup (\mathit{cons}^{\varphi_4}\, y_1\, y_1) \cup \alpha)\, y_2) \cup \tilde{\alpha}_2])\\ y_1 \to^{+} y_2 \cup \tilde{\alpha}_3)&\end{aligned}$$

where we have informally used *rec* rather than μ for recursive types. 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 (Condition 2), and φ_3 and φ_4 because they do not occur negatively in the above type (Condition 3). Replacing the useless variables with $\emptyset$ and **–** as appropriate and replacing the remaining flag variables φ_1, φ_2 with **+** yields

$$\begin{aligned}\forall\alpha.\,(rec\ ([&y_1\quad nil^{+} \cup (cons^{+}\, y_1\, y_1) \cup \alpha]\\ [&y_2\quad nil^{+} \cup (cons^{+}\,(nil^{-} \cup (cons^{-}\, y_1\, y_1) \cup \alpha)\, y_2) \cup \emptyset])\\ &y_1 \rightarrow^{+} y_2 \cup \emptyset).\end{aligned}$$

Changing syntax, we have the presentation type

$$\begin{aligned}(rec\ ([&Y1\ (+\ nil\ (cons\ Y1\ Y1)\ X1)]\\ [&Y2\ (+\ nil\ (cons\ (+\ (not\ nil)\ (not\ cons)\ X1)\ Y2))])\\ &(Y1\ \texttt{->}\ Y2)).\end{aligned}$$

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 the type scheme

$$\forall\varphi_1\varphi_2.\,(true^{\varphi_1} \cup false^{\varphi_2} \cup \emptyset) \rightarrow^{+} (true^{\varphi_1} \cup false^{\varphi_2} \cup \emptyset) \cup \emptyset \tag{7}$$

flag variable φ_1 indicates that this function returns #t only if it is passed #t. Flag variable φ_2 indicates that this function returns #f only if it is passed #f. These dependencies are lost in translating this internal type to the presentation type ((+ *true false*) –> (+ *true false*)). The problem is that an internal type with flag variables shared at different polarities denotes a conjunction of several partitions (or basic types). For example, the above internal type (7) denotes the conjunction

$$\begin{aligned}&((+\ true\ false)\ \texttt{->}\ (+\ true\ false))\\ and\ &(true\ \texttt{->}\ true)\\ and\ &(false\ \texttt{->}\ false)\\ and\ &((+)\ \texttt{->}\ (+)).\end{aligned}$$

(Note that (+) is the empty presentation type corresponding to $\emptyset$.) Cartwright and Fagan [1991] suggest decoding types that share flag variables at different polarities by enumerating all elements of the conjunction not implied by other elements. In this case, type (7) would be printed as

$$(true\ \texttt{->}\ true)\ and\ (false\ \texttt{->}\ false).$$

But this approach quickly becomes unworkable as the number of shared flag variables increases.

Our translation of flag variables shared at different polarities preserves only one element of the conjunction that the type denotes. The other elements of the conjunction are discarded. The translation preserves the element in which all flag variables are substituted to **+**. This element describes the maximum input a procedure can accept and the maximum output it can produce. For instance, we print presentation type ((+ *true false*) –> (+ *true false*)) for the internal type (7). Appendix B shows by means of a denotational semantics of types as ideals that a presentation type resulting from our translation approximates the internal type. In other words, viewing types as upper bounds on sets of values, the presentation type may describe a larger set than the internal type. In our example, presentation type ((+ *true false*) –> (+ *true false*)) denotes a superset of the set that internal type (7) denotes. Thus our translation is imprecise but correct.

Since our presentation types do not precisely describe all possible typings for an expression, we do not have *principal* presentation types. This is a source of imperfection in our illusion of a polymorphic union type system based on presentation types (discussed in Section 2.2). Fortunately, this imperfection does not seem to matter for practical programming.

4. ACCOMMODATING FEATURES OF REAL LANGUAGES

A practical soft type system must address the features of a real programming language. This section adds several such features to our simple soft type system, with emphasis on features of R4RS Scheme.

4.1 Procedure Types of Higher Arity

In Core Scheme, all procedures take exactly one argument. Many programming languages provide procedures that accept different numbers of arguments. Scheme procedures of the form $(\boldsymbol{\lambda}\ (x_1 \ldots x_n)\ e)$ accept n arguments. Scheme procedures of the form $(\boldsymbol{\lambda}\ (x_1 \ldots x_n\ .\ x_r)\ e)$ accept n or more arguments, with arguments beyond the first n packaged as a list and bound to x_r. Certain Scheme primitives also accept trailing optional arguments. To handle procedures of higher arities, we encode Scheme procedure types using *argument lists*.

We imagine Scheme procedures $(\boldsymbol{\lambda}\ (x_1 \ldots x_n\,[.x_r])\ e)$ as taking a single argument list. Whether they are actually implemented this way is immaterial to the type system. Before executing the body e, a procedure disassembles its argument list into the identifiers $x_1 \ldots x_n$ and optionally x_r. Applications $(e_0\ e_1 \ldots e_n)$ implicitly bundle their arguments $e_1 \ldots e_n$ into argument lists. A $\boldsymbol{\lambda}$-expression receives a run-time *arity check* if it may be applied to the wrong number of arguments, i.e., if it may be passed an argument list of the wrong length.

The binary type constructor $(\mathit{arg}\ \cdot\ \cdot)$ and the type constant *noarg* encode argument list types. The presentation type $(\mathit{T1}\ \mathit{T2}\ \mathit{T3}\ \text{->}\ \mathit{T4})$ now abbreviates

$$((\mathit{arg}\ \mathit{T1}\ (\mathit{arg}\ \mathit{T2}\ (\mathit{arg}\ \mathit{T3}\ \mathit{noarg})))\ \text{->}{*}\ \mathit{T4})$$

where $\text{->}{*}$ is the presentation form of the internal constructor $\rightarrow$. For example, the procedure `map` takes two arguments, the first of which is a procedure of one argument. Procedure `map` has type $((\mathit{X1}\ \text{->}\ \mathit{X2})\ (\mathit{list}\ \mathit{X1})\ \text{->}\ \mathit{X2})$, which abbreviates

$$((\mathit{arg}\ ((\mathit{arg}\ \mathit{X1}\ \mathit{noarg})\ \text{->}{*}\ \mathit{X2})\ (\mathit{arg}\ (\mathit{list}\ \mathit{X1})\ \mathit{noarg}))\ \text{->}{*}\,(\mathit{list}\ \mathit{X2})).$$

The types of procedures of unlimited arity use recursive types. For example, $+$ sums zero or more numbers and has type

$$\begin{aligned} + \ : \ & (\mathit{rec}\ ([\mathit{Y1}\ (+\ \mathit{noarg}\ (\mathit{arg}\ \mathit{num}\ \mathit{Y1}))]) \\ & \quad (\mathit{Y1}\ \text{->}{*}\,\mathit{num})). \end{aligned}$$

When translating to presentation types, we abbreviate recursive argument lists as $(\&\mathit{list}\ T)$. Hence $+$ has the more succinct type

$$+\ :\ ((\&\mathit{list}\ \mathit{num})\ \text{->}\ \mathit{num}).$$

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 arity checks usually indicate program errors.

4.2 Assignment

Scheme includes assignment in several forms. Identifier bindings may be changed by **set!**-expressions; the components of pairs may be mutated by the set-car! and set-cdr! primitives; and elements of vectors may be changed by vector-set!. If our type system did not include polymorphic **let**-expressions, incorporating assignment would be easy. But as many authors have noted, naively combining Hindley-Milner polymorphism and assignment leads to an unsound type system [Wright 1995]. Many solutions to this problem have been proposed [Damas 1985; Greiner 1993; Hoang et al. 1993; Leroy 1992a; Leroy and Weis 1991; Talpin and Jouvelot 1992; Tofte 1990; Wright 1992; 1995]. We adapt our own solution [Wright 1995], which is the simplest of all.

To accommodate mutable pairs and vectors, our solution restricts polymorphism to syntactic values. That is, we replace Core Scheme's static type inference rule for **let**-expressions from Figure 3 with the following two inference rules.

$$\frac{A \vdash v_1 : \tau_1 \qquad A[x \mapsto \mathit{Close}(\tau_1, A)] \vdash e_2 : \tau_2}{A \vdash (\mathbf{let}\ ([x\ v_1])\ e_2) : \tau_2} \qquad (\mathbf{letval}_{\vdash})$$

$$\frac{A \vdash e_1 : \tau_1 \qquad A[x \mapsto \tau_1] \vdash e_2 : \tau_2 \qquad e_1 \notin \mathit{Val}}{A \vdash (\mathbf{let}\ ([x\ e_1])\ e_2) : \tau_2} \qquad (\mathbf{letexp}_{\vdash})$$

The first rule assigns a polymorphic type to x when the bound subexpression e_1 is a value. The second rule assigns a type that is not polymorphic to e_1 when e_1 is not a value. With these modified rules, it is easy to establish type soundness for a language that includes mutable pairs and vectors. With correspondingly modified soft typing rules, the various correctness theorems for a soft type system follow.

For assignment to identifiers, Scheme provides a **set!**-expression ($(\mathbf{set!}\ x\ e)$). Since the first position of a **set!**-expression is an identifier, a particular **set!**-expression can assign to only one identifier. Hence identifiers can be classified as assignable and nonassignable according to whether they appear in the first position of a **set!**-expression. To accommodate **set!**-expressions in a soft type system, we further constrain the typing rules for **let**-expressions so that the types of expressions bound to assignable identifiers are not generalized.

$$\frac{A \vdash v_1 : \tau_1 \qquad A[x \mapsto \mathit{Close}(\tau_1, A)] \vdash e_2 : \tau_2 \qquad x \text{ is not assignable}}{A \vdash (\mathbf{let}\ ([x\ v_1])\ e_2) : \tau_2} \qquad (\mathbf{letval}_{\vdash})$$

$$\frac{A \vdash e_1 : \tau_1 \qquad A[x \mapsto \tau_1] \vdash e_2 : \tau_2 \qquad e_1 \notin \mathit{Val} \text{ or } x \text{ is assignable}}{A \vdash (\mathbf{let}\ ([x\ e_1])\ e_2) : \tau_2} \qquad (\mathbf{letexp}_{\vdash})$$

We discuss the general advantages of our method of integrating polymorphism and assignment elsewhere [Wright 1995]. In the context of soft typing, the primary advantage of our solution is simplicity: it requires no changes to the set of types. All of the other solutions add notational complexity to types, which makes reasoning with types difficult. Furthermore, the additional notations introduced by these systems would make decoding types into a simple presentation type language difficult. For example, Tofte's system [1990] would introduce two different kinds of type variables and two different kinds of flag variables. While the two kinds of type variables would pose little problem, decoded components of union types would likely require annotations to indicate the kind of flag variable they possess internally. The extra annotations would amount to displaying the flags themselves.

4.3 First-Class Continuations

Scheme and some dialects of Standard ML provide the ability to access a program's continuation through the use of a *call-with-current-continuation* operator (call/cc) [Clinger and Rees 1991]. This operator provides a powerful form of non-local control that can be used to define exceptions, build backtracking algorithms, schedule multiple threads of control, etc. Harper and Lillibridge [1993] discovered that naively combining first-class continuations with polymorphism leads to an unsound type system, just as naively combining assignment and polymorphism does. Fortunately, the same solution of restricting polymorphism to values works for both assignment and first-class continuations [Wright and Felleisen 1994].

The primitive operator call/cc takes a functional argument and applies it to an abstraction of the current continuation, packaged as a function. In our prototype, call/cc has the type

$$(((X1 \text{ -> } X2) \text{ -> } X1) \text{ -> } X1).$$

A use of call/cc may require a run-time check for either of two reasons: (1) the value to which call/cc is applied (of type $((X1 \text{ -> } X2) \text{ -> } X1)$) is not a procedure of one argument or (2) the continuation obtained (of type $(X1 \text{ -> } X2)$) 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

$$(\boldsymbol{\lambda}\ (\mathsf{v})\ (\mathsf{call/cc}\ (\boldsymbol{\lambda}\ (\mathsf{k})\ (\mathsf{v}\ (\boldsymbol{\lambda}\ (\mathsf{x})\ (\mathsf{k}\ \mathsf{x})))))).$$

This transformation, a composition of three η-expansions, introduces an explicit $\boldsymbol{\lambda}$-expression for the continuation. The expression $(\boldsymbol{\lambda}\ (\mathsf{x})\ (\mathsf{k}\ \mathsf{x}))$ will receive an arity check if the continuation may be mistreated.

4.4 Pattern Matching

Many modern programming languages such as Standard ML, Haskell, and Miranda include pattern-matching expressions. Pattern matching facilitates expressing complicated local control decisions in a concise and readable manner. In ordinary statically typed languages, pattern matching is largely a notational convenience and could be provided as a syntactic abbreviation (macro). But in a soft type system where union types overlap, pattern matching provides an important additional advantage. Pattern matching enables the type checker to "learn" more precise types from expressions that test the types of values.

To illustrate how pattern matching enables more precise type assignment, consider the following expression that tests the type of x:

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

From the expression (**if** e 0 (cons 1 '())), x has type (+ *num* (*pair num nil*)). 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 our soft type system inserts a run-time check at car.

In contrast, the equivalent expression

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

uses pattern matching to test the type of x. The **match**-expression compares the value of x against the *patterns* (a . _) and b. For the first pattern that matches the value of x, the corresponding body is evaluated with any identifiers in the pattern bound to corresponding parts of the value of x. The pattern (a . _) matches a pair, binding a to (car x). The pattern b matches any value, binding b to the entire value of x. This **match**-expression couples the type test to the decomposition of x. By extending the type system to directly type pattern-matching expressions, we can avoid unnecessary run-time checks.

Assigning types to pattern-matching expressions is more difficult in our soft type system than for languages such as ML where types do not overlap. To type a **match**-expression

```
(match e
  [p1 e1]
    ...
  [pn en])
```

we first compute a tidy type for each pattern p_i. We then assemble the pattern types into an input type for the match expression that covers all the pattern types. If the type of the input expression e is larger than the combined pattern types, the **match**-expression requires a run-time check. For instance, the expression

```
(match e
  [() e1]
  [(a . b) e2])
```

has input type (+ *nil* (*cons X1 X2*)). If the type of e includes any types other than *nil* or *cons*, this **match**-expression requires a run-time check.

Because of the restrictions of tidiness, the input type may express a larger type than the set-theoretic union of the pattern types. For example, the least tidy type that is an upper bound of the pattern types (*cons true false*) and (*cons false true*) is

(*cons* (+ *true false*) (+ *true false*)).

Thus a **match**-expression such as

```
(match e
  [(#t . #f) e1]
  [(#f . #t) e2])
```

requires a run-time check even if e has type (*cons* (+ *true false*) (+ *true false*)). We call such **match**-expressions *inexhaustive* because the patterns do not exhaust all possibilities covered by the assembled input type. To find a tidy upper bound of the pattern types, we use a unification algorithm.

Our prototype supports a subset of a general pattern-matching extension that we developed for Scheme [Wright and Duba 1993]. To improve the treatment of ordinary Scheme programs that do not use pattern matching, our prototype translates simple forms of type testing **if**-expressions into equivalent **match**-expressions.

4.5 Data Definition

Scheme provides 11 different kinds of atomic data values: #t, #f, '(), numbers, symbols, strings, characters, input ports, output ports, promises, and an end-of-file object; and three kinds of composite data objects: pairs, vectors, and procedures. Our prototype assigns each of these kinds of data a distinct type. But as these are the only kinds of data objects that Scheme provides, our prototype will infer union and recursive types over only these types.

To facilitate more informative and more precise type assignment, we include an extension to Scheme to permit the definition of new kinds of data. The definition

(**define-structure** ($\kappa\ x_1 \ldots x_n$))

declares a new type constructor κ with arity n. This definition also introduces a value constructor make-κ, a predicate κ?, selectors κ-x_1 ... κ-x_n, and mutators set-κ-x_1! ... set-κ-x_n! with the following types:

make-κ	:	($X1 \ldots Xn$ –> ($\kappa\ X1 \ldots Xn$))
κ?	:	($X1$ –> (+ *true false*))
κ-x_1	:	(($\kappa\ X1 \ldots Xn$) –> $X1$)
$\vdots$		
κ-x_n	:	(($\kappa\ X1 \ldots Xn$) –> Xn)
set-κ-x_1!	:	(($\kappa\ X1 \ldots Xn$) $X1$ –> *void*)
$\vdots$		
set-κ-x_n!	:	(($\kappa\ X1 \ldots Xn$) Xn –> *void*)

A **define-structure** definition can be used at top level or as an internal definition within a **λ**- or **let**-expression. An internal **define-structure** expression is *nongenerative*: repeated invocations of its containing **λ**-expression will construct identical constructor, predicate, selector, and mutator procedures.

Programs that use **define-structure** are assigned more informative and more precise types than those that encode data structures using lists or vectors. Without using **define-structure**, an expression parser for Scheme that represented abstract syntax elements as lists with a distinguishing symbol as the first element ('**λ**, 'app, etc.) would have a type like

(*rec* ([*Y1* (+ *num nil false true char sym str* (*vec Y1*) (*cons Y1 Y1*))])
(*Y1* –> *Y1*)).

Such a type conveys little useful information about the form of output the expression parser produces. But with judicious use of **define-structure**, we can build

```
(rec ([Y1  (+ num nil false true char sym str (vec Y1) (box Y1) (cons Y1 Y1))]
      [Y2  (+ (And (list Y2))
              (App Y2 (list Y2))
              (Begin (list Y2))
              (Const (+ num nil false true char sym str) sym)
              (If Y2 Y2 Y2)
              (Lam (list sym) (Body Y3 (list Y2)))
              (Let (list (Bind sym Y2)) (Body Y3 (list Y2)))
              (Let* (list (Bind sym Y2)) (Body Y3 (list Y2)))
              (Letr (list (Bind sym Y2)) (Body Y3 (list Y2)))
              (Or (list Y2))
              (Prim sym)
              (Delay Y2)
              (Set! sym Y2)
              (Id sym)
              (Vlam (list sym) sym (Body Y3 (list Y2)))
              (Match Y2 Y4)
              (Record (list (Bind sym Y2)))
              (Field sym Y2)
              (Annotation Y1 Y2))]
      [Y3  (list (+ (Define (+ false sym) (box Y2))
                    (Defstruct
                      sym
                      (cons sym Y1)
                      sym
                      sym
                      (list (+ (Some sym) None))
                      (list (+ (Some sym) None))
                      (list (+ false true)))))]
      [Y4  (list (Mclause Y5 (Body Y3 (list Y2)) (+ false sym)))]
      [Y5  (+ (Pconst (+ num char sym str) sym)
              (Pvar sym)
              (Pobj sym (list Y5))
              Pany
              Pelse
              (Pand (list Y5))
              (Ppred sym))])
  (Y1 -> Y2))
```

Fig. 6. Expression parser type.

procedures that have quite informative types. For example, we can represent different abstract syntax elements with different kinds of data:

```
(define-structure (Id name))
(define-structure (Lam args body))
(define-structure (App fun args))
   ⋮
```

With these definitions, Figure 6 shows the type inferred by our prototype for its own expression parser. *Y1* is the input type to the parser, an "s-expression." *Y2* is the output type of the parser, an abstract syntax tree. *Y3* is a list of definitions (either **define** or **define-structure**) that may appear in a **λ**- or **let**-expression. Pattern-matching expressions use a list of match clauses *Y4*, and the first element of each clause is a pattern *Y5*.

As we discussed in Section 4.2, our soft type system restricts polymorphism to syntactic values. Since **define-structure** introduces mutation procedures for constructed values, the values built by defined constructors cannot be considered syntactic values. Hence our prototype also includes an explicit facility for defining new immutable forms of data. The definition

(**define-const-structure** $(\kappa\ x_1 \ldots x_n)$)

declares a new type constructor, a value constructor, a predicate, and selectors, just as **define-structure** does, but no mutators. Applications of a constructor make-κ that was introduced by **define-const-structure** to operands which are syntactic values can be treated as syntactic values. Programs using **define-const-structure** may be assigned more precise types than programs using **define-structure**. (Alternatively, the type system could implicitly treat as immutable any fields of a **define-structure** definition for which the corresponding mutator is never used.)

4.6 Type Annotations

Practical static type systems that support type inference usually include a facility for explicitly specifying the type of an identifier or expression. Explicit type annotations allow the programmer to embed type information in the source code that will be automatically verified when the program is changed. This information is helpful to programmers reading or maintaining a program. Type annotations can also be used to restrict the applicability of a polymorphic procedure by specifying a more specific type.

In a static type system, type annotations can be included by simply adding a new expression form (: τ e) with the following typing rule:

$$\frac{A \vdash e : \tau \qquad \tau \text{ is closed}}{A \vdash (:\ \tau\ e) : \tau}$$

Semantically, an explicit type annotation behaves as an identity function. Its typing rule rejects programs for which the subexpression e does not have the indicated type.

Including explicit type annotations in a soft type system is not as straightforward. The problem is what to do with annotations that are not satisfied. Bearing in mind our desire to ensure safe execution, we see two extreme choices, with hybrid solutions between the two.

At one extreme, we may consider inserting run-time checks at unsatisfied annotations. Unfortunately, this solution is generally infeasible. While it is easy to insert a run-time check for a simple annotation such as (: *num* e), some annotations can require arbitrarily expensive run-time checks. For instance, the annotation (: (*list num*) e) may require an unbounded number of number? tests, as e could be an arbitrarily long list. Inserting run-time checks for higher-order annotations such as (: (*num* –> *num*) e) is impossible without altering the semantics of the program.

At the other extreme, we can treat type annotations the same way as in a static type system. When an expression does not satisfy a type annotation, reject the program. While this solution seems less in keeping with soft typing, it treats all type annotations uniformly, whether they are simple, complex, or higher-order.

Our prototype adopts a hybrid solution. For any annotation that is not satisfied, our type checker inserts an *error* (see Section 2.7). Any program that reaches an unsatisfied annotation will terminate with an error message.

4.7 Generic Arithmetic

Scheme's number system supports arithmetic on representations of small integers, arbitrarily large integers, rational numbers, complex numbers, and inexact numbers. These representations are organized into a natural hierarchy, and the arithmetic operators accept values from all representations in the hierarchy. Each arithmetic operator must dispatch on the representations of its arguments, and these dispatches are often referred to as "type checks." It seems natural to ask whether soft typing can be extended to eliminate these dispatches.

To express the different number representations, we might replace the type constructor *num* with the constructors *smallint*, *bigint*, *rational*, *complex*, and *float*. Constants would then be assigned appropriate types according to their magnitude and syntactic form. Arithmetic operators would use the subtyping provided by flag variables to accept inputs of any numeric type. But the output types of most arithmetic operators would have to be imprecise. Even if called with inputs of type *smallint*, the operator + may yield results in *smallint* or *bigint*. Hence the type assigned to a small integer induction variable of a loop that uses + to increment the induction variable would include *bigint*. This would preclude elimination of the dispatch in +. Consequently, the soft typing framework described here is too weak to effectively eliminate dispatches in generic arithmetic operators.

5. EXPERIENCES WITH A PROTOTYPE SOFT TYPE SYSTEM

In this section, we present some experiences and performance results gathered with Soft Scheme, a prototype implementation of our soft type system for R4RS Scheme. Soft Scheme infers types for Scheme programs and inserts run-time checks by a source-to-source transformation from Scheme to Scheme.

5.1 The Utility of Type Information

We present an example to illustrate the utility of type information for reasoning about Scheme programs and for finding bugs. Our example concerns *boolean formulae* that have the representation

$$b \;::=\; \#\mathsf{t} \;\Big|\; \#\mathsf{f} \;\Big|\; (\boldsymbol{\lambda}\ (x)\ b).$$

We represent a closed boolean formula which is either true or false with Scheme's #t and #f constants. We represent an open formula as a curried procedure that accepts one argument, an assignment for a free variable, and returns a boolean formula. Following are some examples of boolean formulae with possible representations:

true	:	#t
$\neg x$	:	($\boldsymbol{\lambda}$ (x) (**not** x))
$x \wedge y$	:	($\boldsymbol{\lambda}$ (x) ($\boldsymbol{\lambda}$ (y) (**and** x y)))
$x \vee (y \wedge z)$	:	($\boldsymbol{\lambda}$ (x) ($\boldsymbol{\lambda}$ (y) ($\boldsymbol{\lambda}$ (z) (**or** x (**and** y z)))))

A tautology checker is a procedure that accepts a formula and determines whether it is true for all assignments to its free variables. The following procedure is a tautology checker for boolean formulae:

```
(define taut
  (λ (b)
    (match b
      [#t #t]
      [#f #f]
      [_ (and (taut (b #t)) (taut (b #f)))])))
```

For a closed formula, taut returns true or false as appropriate. For an open formula, taut tries both #t and #f as assignments for the free variable and recursively calls itself to test the simplified formula.

Soft Scheme inserts no run-time checks for the following program that exercises taut:

```
(define taut ...)
(define a (taut #t))
(define b (taut not))
(define c (taut (λ (x) (λ (y) (and x y)))))
```

Soft Scheme assigns a, b, and c the type (+ *true false*). The tautology checker itself has type

(*rec* ((*Y1* (+ *true false* ((+ *true false*) –> *Y1*))))
  (*Y1* –> (+ *true false*)))).

That is, taut takes input *Y1* and returns either true or false. The input *Y1* is the type of a boolean formula. *Y1* is either true, false, or a procedure representing an open formula that takes true or false to *Y1*.

The absence of run-time checks in the above program verifies that it will not suffer a run-time failure. While impossibility of failure is certainly reassuring information, large programs usually contain some run-time checks. The types of a program's top-level definitions can also indicate program bugs. For example, consider the following incorrect variation of the above program:

```
(define wrong-taut
  (λ (b)
    (match b
      [#t #t]
      [#f #f]
      [_ (and (wrong-taut (b #t)) (wrong-taut #f))])))
(define a (wrong-taut #t))
(define b (wrong-taut not))
(define c (wrong-taut (λ (x) (λ (y) (and x y)))))
```

This program requires no run-time checks. The applications of wrong-taut all yield the same values for a, b, and c as they do for the correct program. But wrong-taut's type

(*rec* ((*Y1* (+ *true false* (*true* –> *Y1*))))
  (*Y1* –> (+ *true false*)))

indicates that something is amiss. Open formulae should be procedures that accept type (+ *true false*), not just *true*. Applied to (**λ** (x) x), wrong-taut yields #f rather than #t as a tautology checker should.

To further illustrate the utility of type information, consider the following program that uses the correct version of taut:

```
(define taut ...)                                                    (8)
(define d (taut taut))
```

Soft Scheme inserts no run-time checks for this program. The type checker assigns type (+ *true false*) to d. It turns out that taut itself represents the boolean formula x if some of its functionality is ignored (the last clause [_ (**and** ...)] of its definition). Hence taut is a valid input to taut, and the application (taut taut) returns #f. We can see that the application (taut taut) makes sense with an informal understanding of presentation types as sets. Observe that the type of an argument should be a subset of the input type the function expects. In this case, from the presentation type for taut, we should have

$$(\mathit{Y1} \;\texttt{->}\; (+\ \mathit{true}\ \mathit{false})) \;\subseteq\; \mathit{Y1}.$$

Expanding *Y1* we obtain

$$(\mathit{Y1} \;\texttt{->}\; (+\ \mathit{true}\ \mathit{false})) \;\subseteq\; (+\ \mathit{true}\ \mathit{false}\ ((+\ \mathit{true}\ \mathit{false}) \;\texttt{->}\; \mathit{Y1})),$$

which simplifies to

$$(\mathit{Y1} \;\texttt{->}\; (+\ \mathit{true}\ \mathit{false})) \;\subseteq\; ((+\ \mathit{true}\ \mathit{false}) \;\texttt{->}\; \mathit{Y1}).$$

Since a containment $\tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2'$ implies $\tau_2 \subseteq \tau_2'$ and $\tau_1' \subseteq \tau_1$ (recall that $\to$ is contravariant in its first argument), we need only

$$(+\ \mathit{true}\ \mathit{false}) \;\subseteq\; \mathit{Y1}$$

which certainly holds.

The application (taut taut) illustrates the importance of our type-swapping extension for obtaining more precise types (Section 2.6.2). Without this extension, Soft Scheme inserts three run-time checks into program (8) as follows:

```
(define taut
  (λ (b)
    (match b
      [#t #t]
      [#f #f]
      [_ (and (CHECK-ap taut (b #t)) (CHECK-ap taut (b #f)))])))
(CHECK-ap taut taut)
```

These run-time checks indicate that the applications of taut perform unnecessary tests to ensure that their function positions evaluate to procedures. The problem is that without type swapping, the Rémy encoding used in our internal types does not provide the subtyping containment (*Y1* –> (+ *true false*)) ⊆ *Y1*. From the definition of taut alone, the internal type inferred for taut is[9]

$$\begin{array}{l}\forall \varphi_1 \varphi_2 \varphi_3 \alpha_1 \alpha_2 . \\ \quad (rec\ ([Y1\ \ true^{\varphi_1} \cup false^{\varphi_2} \cup ((true^{+} \cup false^{+} \cup \alpha_1) \rightarrow^{\varphi_3} Y1) \cup \tilde{\alpha}_3]) \\ \qquad (Y1 \rightarrow^{+} (true^{+} \cup false^{+} \cup \alpha_2)) \cup \tilde{\alpha}_4).\end{array}$$

Typing the application (taut taut) requires replacing $\tilde{\alpha}_4$ with $true^{+} \cup false^{+} \cup \alpha_5$. From its type, taut now appears to be either true, false, or a procedure. Hence the type checker inserts run-time checks where taut is applied. Since $\tilde{\alpha}_4$ is an absent variable occurring in only a positive position, we can swap it with a fresh type variable. With type swapping, taut has type

$$\begin{array}{l}\forall \varphi_1 \varphi_2 \varphi_3 \alpha_1 \alpha_2 \alpha_6 . \\ \quad (rec\ ([Y1\ \ true^{\varphi_1} \cup false^{\varphi_2} \cup ((true^{+} \cup false^{+} \cup \alpha_1) \rightarrow^{\varphi_3} Y1) \cup \tilde{\alpha}_3]) \\ \qquad (Y1 \rightarrow^{+} (true^{+} \cup false^{+} \cup \alpha_2)) \cup \alpha_6).\end{array}$$

The application (taut taut) does not cause the type checker to insert any unnecessary run-time checks.

5.2 Overcoming the Limitations of Static Typing

The tautology checker illustrates a limitation of statically typed languages that our soft type system overcomes. Our representation of boolean formulae is convenient for reusing existing procedures such as not and taut itself. But this representation confounds conventional static type systems like that of Standard ML because conventional static type systems do not permit mixing boolean and procedural values. To write taut in ML, we must introduce a new datatype `formula`, injections from booleans and procedures into `formula`, and projections out of `formula`. We can simulate the new datatype in Soft Scheme by declaring the variants of the datatype with our data definition facility:

```
(define-const-structure (C _))    (define-const-structure (O _))
(define C make-C)                 (define O make-O)
```

Our intention is to use C to inject a closed formula into the imaginary datatype, and O to inject an open formula into the datatype. Given these definitions, we can write ml-taut and its applications so as to never mix boolean values with procedures:

```
(define ml-taut
  (λ (b) (match b
           [($ C #t) #t]
           [($ C #f) #f]
           [($ O p) (and (ml-taut (p #t)) (ml-taut (p #f)))])))
(define a (ml-taut (C #t)))
(define b (ml-taut (O (λ (x) (C (not x))))))
(define c (ml-taut (O (λ (x) (O (λ (y) (C (and x y))))))))
(define d (ml-taut (O (λ (x) (C (ml-taut (C x)))))))
```

[9]Here we are not using the extension that permits generalizing absent variables. With that extension, $\tilde{\alpha}_3$ and $\tilde{\alpha}_4$ are generalized. The application (taut taut) does not need a run-time check, but two unnecessary run-time checks that can be eliminated by type swapping are still inserted in the body of taut.

```
datatype formula =
    C of bool
  | O of bool -> formula

fun ml_taut (C true)  = true
  | ml_taut (C false) = false
  | ml_taut (O p)     = (ml_taut (p true)) andalso (ml_taut (p false))

val a = ml_taut (C true)
val b = ml_taut (O (fn x => (C (not x))))
val c = ml_taut (O (fn x => (O (fn y => (C (x andalso y))))))
val d = ml_taut (O (fn x => (C (ml_taut (C x)))))
```

Fig. 7. Standard ML version of taut.

The type of ml-taut confirms that booleans and procedures are never mixed:

$$(\mathit{rec}\ ((\mathit{Y1}\ (+\ (O\ ((+\ \mathit{true}\ \mathit{false})\ \text{->}\ \mathit{Y1}))\ (C\ (+\ \mathit{true}\ \mathit{false})))))$$
$$(\mathit{Y1}\ \text{->}\ (+\ \mathit{true}\ \mathit{false}))))$$

Figure 7 presents the Standard ML version of this program. Since the ML program explicitly declares `formula` as a new type where Soft Scheme uses *Y1*, the ML tautology checker has type *formula* –> *bool*.

Observe that in order to make taut pass the ML type checker, we introduced injections and projections both in taut itself and in the data values passed to taut. These injections and projections add both unnecessary semantic complexity and run-time overhead. The difficulty of inserting the necessary injections and projections into data values to be passed to taut impedes reusing existing procedures as boolean formulae. Hence an ML programmer would most likely choose a different representation for boolean formulae than ours.

Cartwright and Felleisen [1994] present a more complex example that illustrates how our soft type system overcomes the limitations of traditional static type systems. They implement an extensible framework for specifying denotational semantics of programming languages as a set of layered interpreter fragments. Each layer provides parsing, evaluation, and printing routines for different language facilities. An interpreter for a complete language is constructed by composing appropriate layers of interpreter fragments. Cartwright and Felleisen have fully implemented their framework in Soft Scheme using parameterized modules for interpreter layers. In contrast, Steele [1994] attempts to use a similar approach in Haskell [Hudak et al. 1992] to compose interpreters from pseudomonads. His program implements interpreter layers as association-lists of functions. While the program is semantically correct and individual layers are typable, Haskell's static type system is unable to type the complete program that composes layers to build an interpreter. Steele wrote a special-purpose program simplifier to simplify the function application that composes interpreter layers such that the simplified program is typable.

5.3 Optimizing Dynamically Typed Programs

Determining the benefit of soft typing to the programming process is difficult. There is no easy way to perform a controlled study of programmer productivity. But several aspects of our prototype are amenable to rigorous measurement and

Program	Size (lines)	Type checking time (seconds)
Tak	23	0.06
Takl	40	0.08
Cpstak	41	0.07
Div	50	0.10
Ctak	55	0.08
Deriv	58	0.21
Destruct	64	0.15
Fft	116	0.19
Dderiv	127	0.27
Traverse	156	0.29
Puzzle	165	0.33
Interp	197	0.37
Lattice	217	0.28
Browse	218	0.37
Check	275	0.82
Takr	520	1.23
Boyer	607	2.54
Graphs	623	0.55
Nbody	873	1.71
Dynamic	2312	3.95
Slatex	2793	3.21
Nucleic	3329	4.91
SoftScheme	5923	18.17

Fig. 8. Type checking times on 120MHz Pentium.

analysis. We can analyze the frequency with which the prototype inserts run-time checks; we can determine the effect of soft typing on execution time by comparing soft typed programs with conventional dynamically typed programs; and we can measure the speed at which our prototype analyzes typical programs.

5.3.1 *Minimizing Run-time Checking.* To investigate the effectiveness of our prototype at minimizing run-time checking, we applied Soft Scheme to a suite of programs ranging in size from a few dozen lines to several thousand. Figure 8 indicates the sizes of the test programs in lines of source code and the times required to type check them under Chez 5.0 on a 120MHz Pentium with 64 megabytes of physical memory. Interp implements an extensible denotational framework [Cartwright and Felleisen 1994]. Unlike the other programs in our benchmark suite, Interp was engineered specifically for Soft Scheme. The purely functional program Lattice enumerates the lattice of maps between two lattices. Check is a simple polymorphic type checker. Graphs counts the number of directed graphs with a distinguished root and k vertices, each having out-degree at most 2. This program makes extensive use of mutation and vectors. N-Body is a Scheme implementation [Zhao 1987] of the Greengard [1987] multipole algorithm for computing gravitational forces on point-masses distributed uniformly in a cube. Dynamic is an implementation of a tagging optimization algorithm [Henglein 1992a] for Scheme. Slatex is a package for typesetting Scheme code.[10] Nucleic is a constraint satisfaction algorithm used to determine the three-dimensional structure of nucleic acids [Feeley et al. 1994].

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

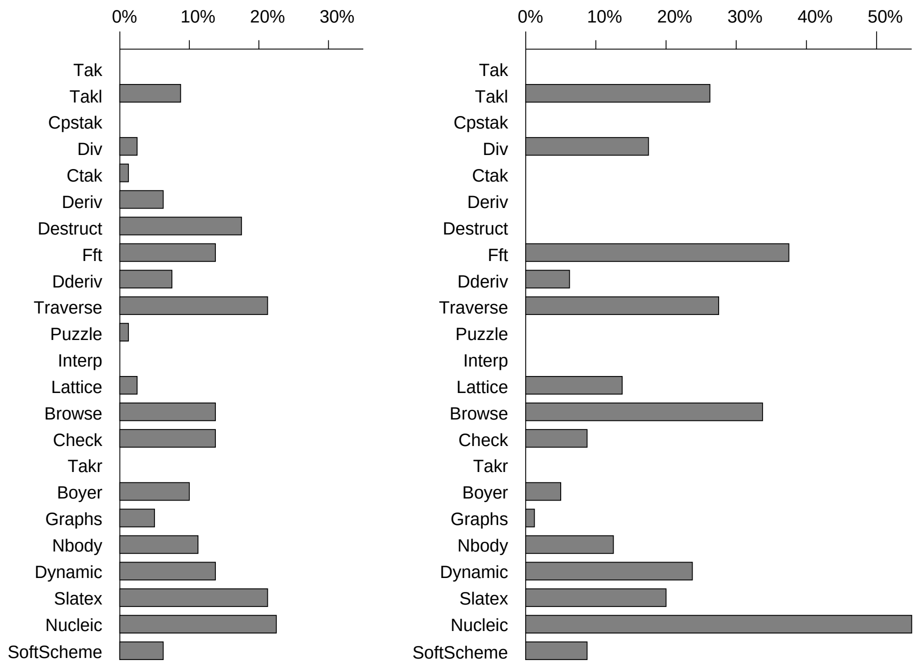

Fig. 9. Static and dynamic frequencies of run-time checks.

It is floating-point intensive and uses an object package implemented using macros and vectors. SoftScheme is our soft type checker soft typing itself. The remaining programs in our test suite are Scheme versions of most of the Gabriel Common Lisp benchmarks.[11]

Figure 9 summarizes run-time checking for our test suite. The percentages indicate how frequently our system inserts run-time checks compared to conventional dynamic typing. The static frequency expressed by the left-hand graph indicates the incidence of run-time checks inserted in the source code. The dynamic frequency expressed by the right-hand graph indicates how often the inserted checks are executed. For example, Soft Scheme places run-time checks at 9% of the potential sites for run-time checks in the Boyer benchmark. Only 5% of the potential sites Boyer encounters during execution involve run-time checks.

Some programs such as Div, Fft, Browse, and Nucleic have a higher dynamic incidence of run-time checks than their static incidence. These programs have run-time checks in frequently executed inner loops. Fft is particularly notable: only 20% of its sites require run-time checks, yet 40% of sites encountered at execution require run-time checks. We examine Fft in detail in the next section.

Some programs such as Destruct, Check, Boyer, and Graphs exhibit a lower dynamic incidence of run-time checks than their static incidence. These programs have run-time checks in less frequently executed code. While more precise type assignment could reduce the static incidence of run-time checks, this reduction is

[11] Ibid.

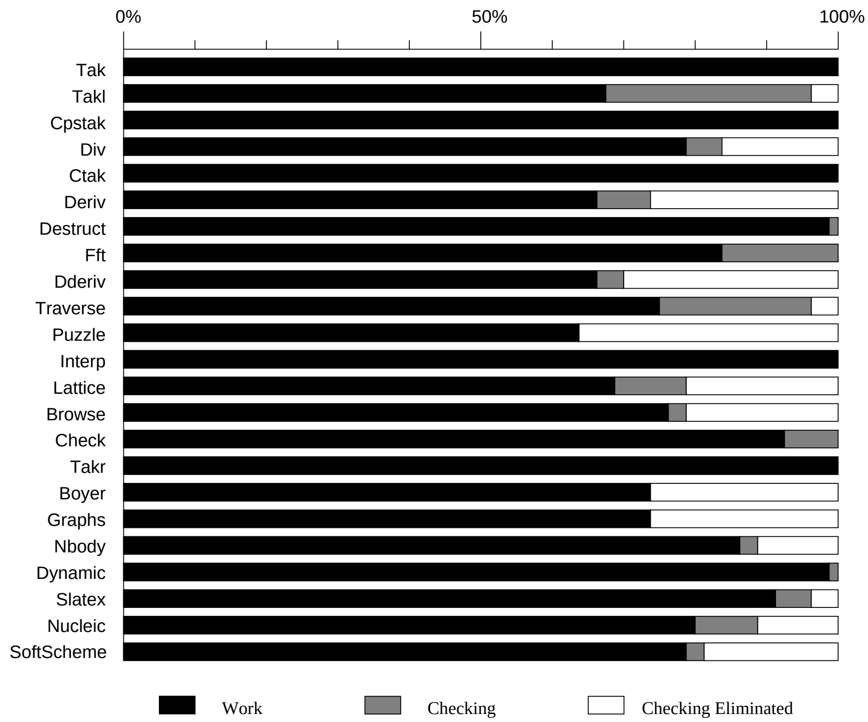

Fig. 10. Execution times for 150MHz Mips R4400.

unlikely to have much effect on the execution times of these programs. At the extreme, Tak, Cpstak, Interp, and Takr have no run-time checks at all because they are statically typable in our underlying static type system.

5.3.2 *Execution Time.* Figures 10 and 11 indicate the breakdown of execution times for our test programs under Chez Scheme 5.0 on a 150MHz Mips R4400 and a 120MHz Pentium. The times are normalized with respect to execution time under Chez Scheme's optimize-level 2, which performs some optimizations but retains run-time checks to ensure safe execution. The black section indicates the fraction of execution time that each program spends doing useful work. This fraction was measured by using optimize-level 3 to turn off all run-time checks. The grey and white sections indicate time spent performing run-time checks. Together, they indicate the time the benchmarks spend performing run-time checks under ordinary dynamic typing, i.e., the difference between optimize-level 2 and optimize-level 3. Since Chez Scheme uses local analysis at optimize-level 2 and higher to safely eliminate some run-time checks, these programs would spend an even greater fraction of execution time performing run-time checks with a naive compiler. The grey section indicates the time each benchmark spends performing run-time checks under soft typing. This fraction was measured by using optimize-level 3 to turn off all run-

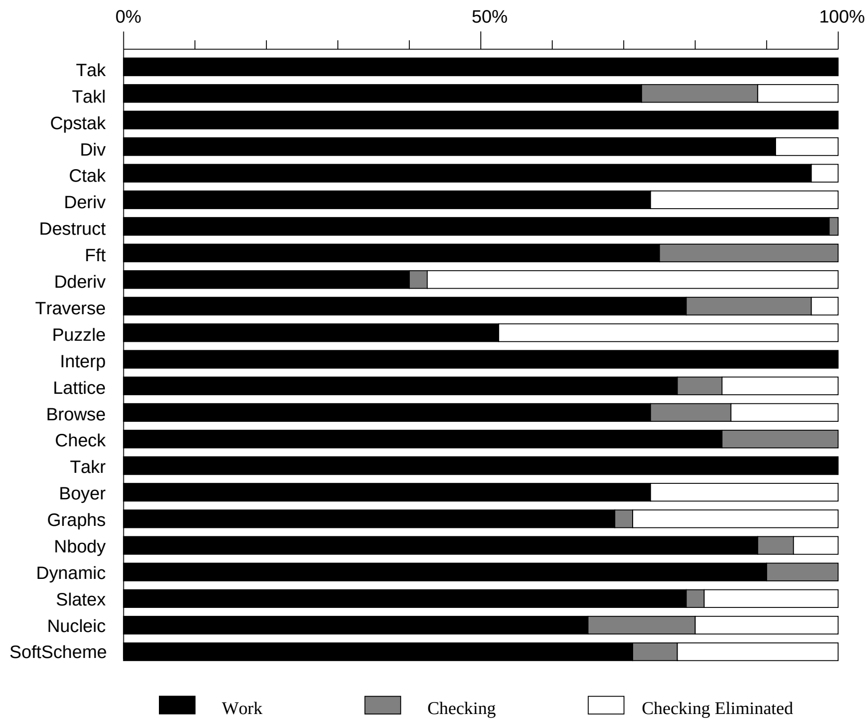

Fig. 11. Execution times for 120MHz Pentium.

time checks other than those explicitly inserted by Soft Scheme.[12] Thus the white section indicates the execution time soft typing saves by eliminating unnecessary run-time checks. On average, this amounts to 11% on the Mips machine and 14% on the Pentium machine.

Soft Scheme significantly decreases time spent performing run-time checks for most of the benchmarks, 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. For example, programs such as Takl, Fft, Traverse, and Check do not expose enough type information to enable the type checker to remove critical run-time checks. Programs such as Tak, Cpstak, Ctak, Destruct, Interp, and Takr spend no appreciable fraction of time performing run-time checks. But Puzzle and Dderiv illustrate the dramatic benefit (speedup by a factor of two or more) that can be obtained when run-time type checks are removed from inner loops.

5.4 From Prototyping to Production

Soft type systems facilitate both the rapid development of software prototypes *and* the evolution of prototypes into robust and efficient production programs. In this section, we illustrate how Soft Scheme may be used to evolve prototypes into

[12] Optimize-level 3 still retains argument count checks and some checks in primitives such as assoc and member.

efficient production programs. We take six programs that exhibit potential for improvement from Figures 10 and 11. Guided by the information produced by Soft Scheme, we apply simple semantics-preserving transformations to rewrite Takl, Div, Fft, Traverse, Slatex, and Nucleic to make them run significantly faster.

5.4.1 *Fft.* Inspecting the run-time checks inserted by Soft Scheme into Fft, we see that all of the checks are vector operations applied to the variables ar and ai, such as (CHECK-vector-ref ai i). The checks are necessary because both ar and ai have type (+ *num* (*vec num*)). In the Fft source, we find the following code fragment:

```
(define fft
  (λ (areal aimag)
    (let ([ar 0] [ai 0])
      (set! ar areal)
      (set! ai aimag)
      ⋮
```

The arguments areal and aimag to fft are always vectors. The variables ar and ai are immediately assigned these vectors and never reassigned. Hence *num* in the types of ar and ai stems only from their useless initial value 0. By simplifying the code as follows:

```
(define fft
  (λ (areal aimag)
    (let ([ar areal] [ai aimag])
      ⋮
```

our type checker is able to assign type (*vec num*) to both ar and ai. Soft Scheme inserts no run-time checks at all in the modified program.

5.4.2 *Div.* Div contains a procedure that constructs lists of length n:

```
(define create-n
   (λ (n)
     (let loop ([n n][a '()])
       (if (= n 0)
           a
           (loop (− n 1) (cons '() a))))))
```

Soft Scheme infers type (*num* –> (*list nil*)) for this procedure. Two similar procedures consume the lists created by create-n, and each of these consumers contains a run-time check at cddr. The code for one of the consumers follows:

```
(define recursive-div2
  (λ (l)
    (if (null? l)
        '()
        (cons (car l) (recursive-div2 (CHECK-cddr l))))))
```

This procedure clearly consumes only even-length lists. The same is true of the other consumer. But the result type of create-n includes both odd- and even-length lists, so the consumers require run-time checks at cddr.

If we unroll the loop in create-n as follows

```
(define create-n-even
  (λ (n)
    (let loop ([n n][a '()])
      (if (= n 0)
          a
          (loop (− n 2) (cons '() (cons '() a)))))))
```

our type checker infers the type

(rec ((Y1 (+ nil (cons nil (cons nil Y1))))) (num −> Y1))

for create-n-even. This type expresses the fact that the constructed lists are of even length. When we replace create-n with create-n-even, the consumers that assume their inputs are even length lists no longer require run-time checks.

5.4.3 *Takl.* Takl contains the following routine to determine whether one list is shorter than another:

```
(define (shorterp x y)
  (and (not (null? y)) (or (null? x) (shorterp (cdr x) (cdr y)))))
```

For this routine, our prototype infers the following imprecise type:

(rec ((Y1 (cons X1 Y1)) (Y2 (cons X2 Y2)))
(Y1 Y2 −> (+ false true)))

The problem is that our simple heuristic to transform **if**-expressions to **match**-expressions does not work for the **and**- and **or**-expressions in this code.

Rewriting shorterp in a different style[13]

```
(define (shorterp x y)
  (cond [(null? y) #f]
        [(null? x) #t]
        [else (shorterp (cdr x) (cdr y))]))
```

enables our prototype to infer a more precise type for shorterp:

((list X1) (list X2) −> (+ false true))

This transformation eliminates two of the five run-time checks from Takl. The remaining run-time checks cannot be safely eliminated because proving their safety requires sophisticated reasoning about the lengths of lists.

5.4.4 *Traverse, Slatex, and Nucleic.* Traverse, Slatex, and Nucleic are larger programs that construct complicated data structures. They use vectors to encode these data structures, since they are written in R4RS Scheme. As Section 4.5 discusses, our prototype assigns rather imprecise types to programs written in this manner.

By carefully rewriting these programs to use **define-structure**, we can achieve more precise type assignment and many fewer run-time checks. The resulting code is also clearer and more abstract than the original. The sizes of these programs preclude discussion of the specific transformations in this article.

[13] **cond** is a macro that expands to a series of **if**-expressions.

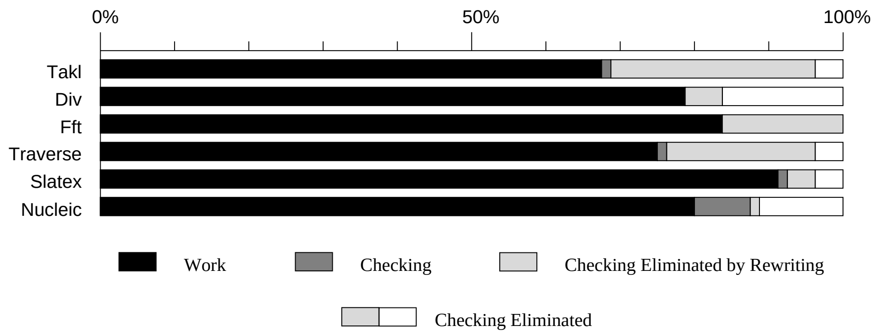

Fig. 12. Improved execution times for 150MHz Mips R4400.

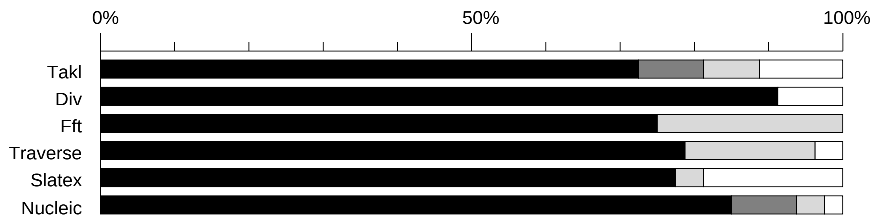

Fig. 13. Improved execution times for 120MHz Pentium.

5.4.5 *Improved Execution Times.* Figures 12 and 13 present the improved execution times of the modified test programs.[14] The light grey region indicates execution time that the original program spent performing run-time checks, but which is eliminated in the modified program. The dark grey region indicates time that the modified program still spends in run-time checks. In most cases, we were able to rewrite the programs to eliminate the majority of run-time-checking overhead.

5.5 Problems

Soft Scheme represents the first attempt to construct a practical soft type system. While we believe our experiment has been largely successful, there are several aspects of our prototype that could be improved through either better soft typing technology or by changing the programming language.

5.5.1 *Precision.* As the problem of assigning precise static types to programs is undecidable, any type system must yield conservatively approximate types.[15] We believe that the intuitive models which programmers use to reason about types take this approximation into account. But we have identified several problems with our system that result in less precise types than our intuition leads us to expect.

[14]We were unable to make our version of Nucleic run faster than the original program because the original source uses macros to inline vector-ref operations, while our more abstract version of Nucleic makes procedure calls to structure accessors. The overhead of these procedure calls is quite high. Hence these results compare our version of Nucleic against a modified version of the original that makes procedure calls to vector-ref rather than using macros.

[15]A *precise* type for a procedure such as add1 would completely specify its output for every possible input, i.e., (add1 0) = 1, (add1 1) = 2, etc.

Type Representation. Our tidy union types can express most common Scheme types with sufficient precision. But our system is unable to assign satisfactory types to four of the R4RS Scheme library procedures. 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. Soft Scheme assigns imprecise types to these procedures that sometimes result in unnecessary run-time checks.

For map and for-each, which can apply an N-ary function to N lists, our internal types cannot represent the requirement that the number of lists match the arity of the function. That is, a precise type for map would be

$$\begin{array}{ll} & ((\mathit{X1} \text{ -> } \mathit{X0})\ (\mathit{list}\ \mathit{X1}) \text{ -> } \mathit{X0}) \\ \mathit{and} & ((\mathit{X1}\ \mathit{X2} \text{ -> } \mathit{X0})\ (\mathit{list}\ \mathit{X1})\ (\mathit{list}\ \mathit{X2}) \text{ -> } \mathit{X0}) \\ \mathit{and} & ((\mathit{X1}\ \mathit{X2}\ \mathit{X3} \text{ -> } \mathit{X0})\ (\mathit{list}\ \mathit{X1})\ (\mathit{list}\ \mathit{X2})\ (\mathit{list}\ \mathit{X3}) \text{ -> } \mathit{X0}) \\ \mathit{and} & \quad \ldots \end{array}$$

Dzeng and Haynes [1994] adapt Rémy's encoding technique to give a precise type to variable-arity functions such as Scheme's map. It seems straightforward to extend our soft type system to include their encoding. But as their system relies on an explicit type annotation and does not *infer* this most precise type when given a definition of map, it is not clear whether extending Soft Scheme to include their encoding would be worthwhile. We have found that simply defining a family of map-k functions, one for each arity, is an adequate solution for practical programming.

The function apply takes two or more arguments. The first is a function to apply; the last is a list of argument values; and any intervening arguments are additional individual argument values, i.e.,

$$(\mathsf{apply}\ f\ a_1 \ldots a_n\ l) = (\mathsf{apply}\ f\ (\mathsf{cons}\ a_1 \ldots (\mathsf{cons}\ a_n\ l)\ldots)).$$

Our method of typing variable-arity procedures has directional bias. All elements in the tail of a variable-length argument list must have the same type, as the tail of an argument list is represented by a recursive type. Hence Soft Scheme assigns to apply the internal type

$$((\mathit{arg}^{\varphi_1}\ ((\mathit{arglist}\ \alpha_1) \text{->*}^{\varphi_2} \alpha_2)\ (\mathit{arg}^{\varphi_3}\ (\mathit{list}\ \alpha_1)\ \mathit{noarg})) \text{ ->*}^{+}\ \alpha_2) \cup \emptyset,$$

where $(\mathit{arglist}\ \alpha_1)$ and $(\mathit{list}\ \alpha_1)$ are abbreviations for recursive types. This type causes apply to receive a run-time check if it is passed any individual arguments $a_1 \ldots a_n$, although the application succeeds. Provided that apply is not used in a higher-order manner, this run-time check is easily eliminated by transforming the application as indicated above.

The problem with apply could be solved by making pairs immutable. Recall that argument lists are encoded using the type constructors *arg* and *noarg*. The mutability of *cons* prevents its use for argument lists because type constructors for mutable values must be treated differently from type constructors for immutable values in algorithms such as type swapping (Section 2.6.2). If pairs were immutable, we could assign a reasonable type to apply by eliminating the type constructors *arg* and *noarg* and using the type constructors *cons* and *nil* for argument lists. The internal type for apply would be

$$((\mathit{cons}^{\varphi_1}\ (\alpha_1 \text{->*}^{\varphi_2} \alpha_2)\ \alpha_1) \text{ ->*}^{+}\ \alpha_2) \cup \emptyset, \tag{9}$$

where α_1 matches the entire *argument list* of the passed function f to the remainder of the argument list passed to apply.

The function append suffers a similar problem to apply. It takes a variable number of arguments, the last of which is special:

(append $l_1 \ldots l_n$ v).

Given lists $l_1 \ldots l_n$ and an arbitrary value v, append appends the lists together, placing v in the tail of the list (where v need not be a list). Soft Scheme assigns append the internal type

$((\mathit{arglist}\ (\mathit{list}\ \alpha)) \mathbin{{-}{>}{*}^{+}} (\mathit{list}\ \alpha)) \cup \emptyset,$

which forces the last argument v to have a list type. When v is not a list, an unnecessary run-time check results. Unfortunately, we have no solution for this case.

The difficulties with representing types for certain Scheme functions entices one to consider a more expressive type system, such as that of Aiken et al. (see Section 6.1.2). But aside from the algorithmic inefficiency of systems like theirs, more complicated types will be more difficult for programmers to understand. Overall, we feel that our type representation 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 (see rule **if** in Figure 3). Applications of **λ**-expressions require the types of arguments to match the types the function expects (see rules **ap** and **Cap** in Figure 3). Consequently, type information can flow counter to the direction of value flow. This *reverse flow* of type information results in imprecise types and unnecessary run-time checks. To illustrate reverse flow, consider the program

```
((λ (f y)
   (f y)
   (add1 y))
 (λ (x) (if x x #f))
 1)
```

Reverse flow at the **if**-expression forces the type of x to include *false*. Reverse flow again where the expression (**λ** (x) (**if** x x #f)) is passed to f forces f to have the type $((+\ \mathit{false}\ T) \mathbin{->} (+\ \mathit{false}\ T))$ for some as yet unspecified type T (which turns out to be *num*). Yet again, reverse flow at the application (f y) forces y to have type $(+\ \mathit{false}\ T)$. The inclusion of *false* in the type of y forces a run-time check at the application (add1 y), despite the fact that y is never bound to #f.

Reverse flow arises when the modicum of subtyping provided by our adaptation of Rémy encoding fails. In the above example, the encoding fails to provide subtyping because f is not polymorphic. After rewriting the example as follows

```
(let ([f (λ (x) (if x x #f))])
  ((λ (y)
     (f y)
     (add1 y))
   1))
```

f is assigned the polymorphic internal type

$$\forall\alpha.\,((\mathit{false}^{+} \cup \alpha) \;{-}{>}^{+}\; (\mathit{false}^{+} \cup \alpha)) \cup \emptyset.$$

Type swapping (see Section 2.6.2) replaces this type with the internal type

$$\forall\varphi\alpha.\,((\mathit{false}^{\varphi} \cup \alpha) \;{-}{>}^{+}\; (\mathit{false}^{+} \cup \alpha)) \cup \emptyset,$$

which provides subtyping on the input to f. Hence no reverse flow occurs at the application (f y), and no unnecessary run-time check is inserted at (add1 y).

Addressing the reverse-flow problem requires more sophisticated type inference than unification-based algorithms yield. To this end, we have investigated several adaptations of structural subtyping [Kaes 1992; Mitchell 1991]. Structural subtyping permits subtyping at all function applications and handles **if**-expressions more precisely. By permitting more subtyping, a soft type system based on structural subtyping could 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 consume exorbitant amounts of memory and execution time for even small examples. The problem is that the known techniques for implementing structural subtyping do not preserve sharing between representations of the same type. This sharing is crucial to the efficient implementation of unification-based type inference algorithms [Rémy 1992], particularly in the presence of recursive types.

Other possible solutions to the reverse-flow problem are type inference methods based on constraint solving systems such as those of Aiken et al. [1994] and Heintze [1993]. Section 6 discusses these systems in more detail.

Assignment. Because assignment interferes with polymorphism, and therefore with subtyping, assignment can be a major source of imprecise types (see Section 4.2). Scheme includes both assignable identifiers and mutable data structures.

Assignments to local identifiers seldom cause imprecise types. We postulate two reasons for this. First, identifiers cannot alias with other identifiers or data structures; hence an assignment to a particular identifier only affects the typing of that identifier. Second, locally bound identifiers are seldom used in a way that requires polymorphism or subtyping.

Assignments to global identifiers are more problematic. Since assignable global identifiers are not polymorphic, subtyping does not apply at uses of such identifiers. Hence these identifiers can accumulate large, imprecise types. Furthermore, the program development environments usually associated with Scheme assume that all global identifiers are assignable. As it accepts only complete programs, our prototype can permit subtyping for global identifiers that are not assignable. But we see no way of limiting the types of assignable global identifiers other than by carefully engineering programs to use such identifiers in a disciplined way that promotes precise type assignment.

In R4RS Scheme, pairs and vectors are the only forms of data structures. Both are mutable. In the absence of sophisticated analysis, we must assume that any pair or vector can alias with any other pair or vector. This limits the precision with which ordinary Scheme programs can be typed. Soft Scheme provides the syntactic form **define-const-structure** to enable defining immutable data structures that can be typed more precisely. To reduce the potential for imprecise typing, we would prefer to make Scheme's pairs immutable. Programmers would then be required to explicitly construct reference cells where mutable values are required. This is the solution that Standard ML adopted and has proven practical for accommodating both polymorphism and assignment in the same language. As a pragmatic hack to improve typing of existing Scheme programs, our prototype assumes that pairs are immutable if the mutation procedures set-car! and set-cdr! do not appear in the program.

5.5.2 *Type Size.* While our presentation types are concise and easy to understand, especially in comparison with types such as those that Aiken et al.'s [1994] system infers, the types our system infers for procedures can still be quite large. Figure 6 illustrates a conceptually simple type that occupies an entire page.

Ordinary static type systems solve this problem in two ways. Both solutions could be adapted to a soft type system. The first solution involves introducing abbreviations for commonly used types. In its full generality, this solution requires detecting subgraphs of a type that are isomorphic to the expansion of some abbreviation. The routine that prints types detects such subgraphs and contracts them to the abbreviated name. Soft Scheme does this for the fixed set of abbreviations *list*, *arglist*, and $\rightarrow$. It should be feasible to permit user-defined type abbreviations, provided that the number of abbreviations is fairly small.

The second solution to keeping types small involves introducing a new type name when a new datatype is declared. The Standard ML declaration

```
datatype metaval = Int of int
                 | List of metaval list
                 | Closure of metaval -> metaval
```

introduces constructors named Int, List, and Closure. All of these constructors yield values of type *metaval*. But adapting this solution to soft typing invokes the problems associated with type annotations (see Section 4.6). The constructor Int can only be applied to values of type *int*; the constructor List can only be applied to lists of *metaval*; and the constructor Closure can only be applied to functions from *metaval* to *metaval*. The run-time checks required for applications of constructors such as List can be very expensive, while the run-time checks required for applications of constructors such as Closure are uncomputable.

5.5.3 *Explaining Run-time Checks.* As Tofte [1990, p. 33] and others have observed, with a static type system one must understand *why* a program fails to type check in order to repair it. A similar situation exists for programming with a soft type system. While soft type systems do not reject programs, it is still necessary for programmers to be able to predict the type system. Understanding why a run-time check has been inserted or why a procedure has a particular type is essential to writing programs that have good syntactic type assignment and few run-time checks.

Our type system is somewhat more difficult to predict than the Hindley-Milner system. In part, this is because our system includes extra features—union types and recursive types—that the Hindley-Milner system does not have. The encoding that our system uses to reduce subtyping to polymorphism also contributes to the problem of predicting the type system. Subtyping does not always occur when it "should." Our method of handling assignment by generalizing only syntactic values makes it fairly easy to determine when a procedure will *not* be assigned a polymorphic type and hence when subtyping will *not* apply. But flag variables may fail to be generalized even for syntactic values (because they are free in the type environment). Converting internal types to presentation types can obscure the fact that a flag variable is not generalized.

Initial experimentation with Soft Scheme seems to indicate that the type system can be predicted reasonably well at the level of presentation types. Our prototype includes a crude facility to help find the cause of a run-time check, and as a last resort, the prototype can display internal types.

5.6 Directions for Future Work

So far, our research has concentrated on designing a practical soft type system to insert run-time checks into complete programs. Several promising directions for future research lie in finding other uses for the type information inferred by a soft type system and in analyzing incomplete programs.

5.6.1 *Applications of Type Information.* Soft Scheme uses type information only to optimize the placement of run-time checks. We expect that there are several other important uses for this type information.

Static type systems often couple types to data representations. By doing so, different types of data can be given specialized representations that the underlying hardware may be able to manipulate more efficiently. For example, integers may be represented as 32-bit twos-complement values, while real numbers are represented as 64-bit floating-point values. In our type system, values whose union type contains only one component are never mixed with values of a different type, except through polymorphic functions. A technique similar to Leroy's [1992b] method for unboxing values in ML could permit unboxing values of singular union types. Unboxing certain kinds of values could also simplify the problem of integrating Soft Scheme with other languages, such as C.

We also believe that type information such as Soft Scheme infers has the potential to guide many kinds of program-global optimizations. Various Lisp compilers [Beer 1987; Kaplan and Ullman 1980; Ma and Kessler 1990; MacLachlan 1992] and some object-oriented systems [Chambers 1992; Kind and Friedrich 1993] use iterative flow analysis techniques to determine type information for optimization. Only recently has the idea of using type information derived from type inference for optimization received much attention in the literature.

5.6.2 *Applications to Static Type Checking.* To design a static type system, one partitions the data domain into disjoint *datatypes*, just as we discussed for a soft type system in Section 2.2. But as static type systems without subtyping or union types insist that each expression be assigned a single datatype, the partitions must be larger to yield a useful language. For instance, a single partition or type *list* must include both nil and pairs to permit useful manipulations of lists. Larger partitions mean that more operations include implicit run-time checks even when their arguments have the correct type. With lists, the operations car and cdr, which have types ((*list X1*) -> (*list X1*)) and ((*list X1*) -> *X1*) respectively, must include implicit run-time checks to ensure that their argument is not nil.

A dynamic type system can be seen as a special case of a static type system with only one datatype. Hence soft typing should be applicable to the individual datatypes of a statically typed language. It should be feasible to adapt our method of inferring type information to determine type information for variants in languages such as Standard ML. This type information could be used to eliminate variant checks and suppress unnecessary warning messages corresponding to the eliminated variant checks.

5.6.3 *Programming Environment Issues.* A programming environment should provide support for separate compilation and interactive program development. These are really different facets of the same problem: how to incrementally update information about a program that was obtained through costly analysis. This analysis information may include type information, intermediate code, or even machine code and linkage information.

A complete compiler for a language that includes a soft type system would consist of several phases: type assignment, run-time check insertion, optimization, and code generation. Modifying our type assignment method to compute type information incrementally should be straightforward. We envision using a technique related to but simpler than one that Shao and Appel [1993] proposed for separately compiling ML. Types are inferred independently for separate program modules or definitions. Where a module refers to identifiers from some other module, each external reference is assigned a fresh type variable. After type inference for the module is completed, the types of the external references represent the minimum requirements the module makes of these identifiers. Type information for a complete program is assembled from the types of individual modules by a minor variation on the usual type inference algorithm. Each external reference type is unified with the type of the corresponding definition, or with an instance of the definition's type if the definition is polymorphic.

Modifying the phase that inserts run-time checks to be incremental is more difficult. Whether a primitive within a module requires a run-time check may depend on how that primitive is used. But some run-time checks can be determined to be unnecessary regardless of how the module is used. For example, in the definition

```
(define f
  (λ (x)
    (if (number? x)
        (add1 x)
        0)))
```

the primitive add1 does not require a run-time check regardless of how f is used. This is easy to detect from the fact that the absent type variable in the input to add1 does not appear in the type of f. In general, an incremental run-time check insertion algorithm could determine for each primitive that it

(1) needs a run-time check,

(2) does not need a run-time check, or

(3) may need a run-time check, depending on how its containing module is used.

In the third case, deciding whether the run-time check is necessary requires type information for the complete program.

Incremental optimization and code generation for a language that includes soft typing are mildly more difficult. These phases must accommodate the fact that run-time checks may become necessary or unnecessary as incremental changes are made to the program.

6. RELATED WORK

In this section we place our work in the context of other work on soft typing, optimization, and static type systems.

6.1 Soft Type Systems

While our soft type system is the first practical soft type system to be designed for a realistic programming language, several other soft type systems have been developed for idealized programming languages. Our work began as an attempt to implement and improve a soft type system developed by Cartwright and Fagan. Coincidentally with our work, Aiken, Wimmers, and Lakshman developed a soft type system for the functional language FL.

6.1.1 *Cartwright and Fagan.* Our practical soft type system is based on a soft type system designed by Cartwright and Fagan [1991] and Fagan [1990] for an idealized functional language. 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 early encoding technique Rémy [1989] developed to reduce record subtyping to polymorphism. In Cartwright and Fagan's system (henceforth called *CF*), types consist of either a type variable or a union that enumerates every available type constructor. That is, *CF* types are defined as

$$\tau \;::=\; \alpha \;\Big|\; \kappa_1^{f_1}\vec{\tau}_1 \cup \ldots \cup \kappa_n^{f_n}\vec{\tau}_n \qquad (\mathit{CF\ Types})$$

where $\{\kappa_1, \ldots, \kappa_n\}$ is the set of all type constructors.

Since a type variable cannot appear in a union with type constructors in a *CF* type, one of our types such as

$$((\mathit{false}^{+} \cup \alpha) \to^{+} (\mathit{false}^{-} \cup \alpha)) \cup \emptyset$$

must be represented in *CF* by enumerating all other type constructors in place of α:

$$\begin{array}{l}(\mathit{false}^{+} \cup \mathit{num}^{\varphi_2} \cup \ldots \cup (\mathit{cons}^{\varphi_{n-1}}\, \alpha_{n-1}\, \alpha'_{n-1}) \cup (\alpha_n \to^{\varphi_n} \alpha'_n)) \to^{+} \\ (\mathit{false}^{-} \cup \mathit{num}^{\varphi_2} \cup \ldots \cup (\mathit{cons}^{\varphi_{n-1}}\, \alpha_{n-1}\, \alpha'_{n-1}) \cup (\alpha_n \to^{\varphi_n} \alpha'_n)).\end{array}$$

Hence some types have a much larger representation in Cartwright and Fagan's system than they do in our system. These larger types make type inference much less efficient than in our system. Moreover, the larger types do not have natural decodings into presentation types. Finally, the *CF* representation does not support incremental definition of new type constructors because union types must enumerate all constructors.

Aside from the representation of types, our soft type system differs significantly from Cartwright and Fagan's soft type system in several other ways.

(1) Our system presents types to programmers in a simple presentation type language that is easy to interpret.

(2) Our system addresses various features of realistic programming languages (see Section 4), such as assignment, first-class continuations, pattern matching, data definition, and type annotations. Cartwright and Fagan's system deals with only a functional language with a simple case statement.

(3) We use an operational approach rather than a denotational approach to specifying the language semantics. As a result our theorems are simpler to prove, and our theory can incorporate imperative features such as assignment and continuations. At present, we are not aware of any established techniques based on denotational semantics for proving type soundness for imperative languages.

(4) Finally, we have implemented our system for a realistic programming language. We have therefore been able to evaluate the practicality of soft typing as a tool to improve programming productivity (see Section 5).

6.1.2 *Aiken, Wimmers, and Lakshman.* Aiken et al. [1994] and Aiken and Wimmers [1993] recently developed a sophisticated soft type system for the functional language FL. Their system supports a rich type language that includes union types, recursive types, intersection types, conditional types, and subtype constraints. Their type inference method is based on a procedure for solving type constraints of the form $\tau_1 \subseteq \tau_2$ by reducing compound constraints to simpler ones. Constraints are generated by applications. For example, where e_1 has type τ_1 and e_2 has type τ_2, the application $(e_1\ e_2)$ generates the constraints $\tau_1 \subseteq \tau_3 \rightarrow \tau_4$ and $\tau_2 \subseteq \tau_3$. Their theory provides for inserting run-time checks at primitive operations whose input constraints are not satisfied.

While it seems clear that Aiken et al.'s 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 significantly slower than ours. 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 implemented with acceptable performance, we believe that it could serve as a good basis for a stronger soft type system for Scheme.

6.2 Systems for Optimizing Tagging and Checking

A number of approaches for optimizing tagging and checking in dynamically typed languages have been developed. More recent efforts are based on type systems. Earlier methods were based on flow analysis techniques.

6.2.1 *Systems with a Maximal Type.* Several researchers have developed static type systems that extend a static type discipline with a maximal type $\top$, which is assigned to phrases that are otherwise untypable. These systems insert tagging and checking *coercions* to ensure type safety. Henglein's [1992b] system for optimizing run-time tagging and checking inserts only atomic coercions such as $num \rightsquigarrow \top$ and $\top \rightsquigarrow num$ that are small, constant time operations. Systems proposed by Gomard

[1990] and Thatte [1988; 1990] and studied by O'Keefe and Wand [1992], as well as a more general system proposed by Henglein [1992a], insert compound coercions such as (*list num*) $\rightsquigarrow \top$ and $\top \rightsquigarrow$ (*list num*) that may be much more expensive than atomic coercions.

These systems are designed primarily to optimize tagging and checking, but not to determine useful type information for programmers. Henglein has implemented a prototype of his system that inserts tagging and checking operations into Scheme programs. He reports encouraging results with eliminating unnecessary tagging and checking. But as his prototype assigns the type (*list* T) to both pairs and the empty list, his prototype does not remove run-time checks at car or cdr. Nevertheless, to investigate whether Henglein's system or one like it might be suitable for soft typing, we adapted Henglein's prototype to report the types of all identifiers defined by a program. Running the prototype on its own source code, we found that 50% of all globally defined identifiers had type $\top$ or $\top \rightarrow \top$. Fully 95% of all locally defined names had one of these uninformative types. As Henglein's prototype is a fairly typical Scheme program, it appears that the types Henglein's system assigns are not sufficiently informative to meet the goals of soft typing.

While systems with a maximal type may be able to type all Scheme programs, they provide little information of use to the programmer about the shapes of data structures or the behaviors of procedures. Furthermore, for Gomard's and Thatte's systems, it is not clear how to construct a semantics for programs written by the user. It appears that the programmer must understand the coercion insertion algorithm to understand the meaning of a program.

6.2.2 *Systems with Union Types.* Henglein and Rehof [1995] extended Henglein's system for optimizing run-time tagging and checking to include polymorphism and union types like ours. They use their system to translate Scheme programs to typable ML programs, injecting values tagged into an ML datatype with variants for the different Scheme types. By inferring both tag checking and tagging coercions, their system permits using untagged data representations (uninjected ML values). Coercions need not be placed only at primitive operations, and the system may add coercion parameters to procedures. By adding enough coercion parameters to a polymorphic procedure that its safety can be ensured in any context, the system supports separate compilation. But inserting checking coercions at locations other than primitive operations can change the semantics of a program. Furthermore, it seems likely that passing many extra coercion parameters will have a significant run-time penalty.

Aiken and Fahndrich [1995] show how to use set constraints to infer tagging and checking coercions. Their system subsumes Henglein's original system and is quite similar to Henglein and Rehof's system. Since type inference is performed by solving set constraints, this work suggests the possibility of using more powerful constraint-based systems to infer tagging and checking coercions.

6.2.3 *Flow Analysis.* The designers of optimizing compilers for Scheme and Lisp have developed type recovery procedures based on iterative flow analysis [Beer 1987; Kaplan and Ullman 1980; Ma and Kessler 1990; MacLachlan 1992]. Some object-oriented languages use similar methods [Chambers 1992; Kind and Friedrich 1993]. The information gathered by these systems is important for program optimization, but it is generally too coarse to serve as the basis for a soft type system. Few of the systems infer polymorphic types, and none accommodate higher-order functions in an accurate manner. Most infer types that are simple unions of type constants and constructions. And regrettably, we have been unable to find any precise formal definitions of these kinds of systems.

Shivers [1991] developed a family of techniques based on abstract interpretation to perform flow analysis for Scheme-like languages. His techniques accommodate higher-order functions and can be used to perform type recovery. The simplest technique (0CFA) is too imprecise to yield useful types. For a technique suitable for type recovery (1CFA), Shivers reports times of several seconds under interpreted T (a dialect of Scheme) to infer type information for procedures of less than 20 lines. However, to infer types for an entire program, the type recovery algorithm analyzes each procedure with respect to each call to that procedure [Shivers 1991, p. 121]. Hence this algorithm is likely to be impractical for large programs that make extensive use of higher-order functions.

Heintze [1993] developed a flow analysis method for higher-order languages based on ignoring dependencies between identifiers that is essentially equivalent to 0CFA [Jagannathan and Weeks 1994]. He reports reasonable execution times to analyze ML programs of several thousand lines. With polyvariance extensions to increase precision, we believe this analysis can be used to perform useful type recovery. Flanagan and Felleisen [1995] have implemented a soft type system for Scheme based on this analysis and are currently investigating its effectiveness.

Jagannathan and Weeks [1994] developed a parameterizable flow analysis framework based on 0CFA. Recently, Jagannathan and Wright [1995] implemented a specific instance of this framework using a technique called *polymorphic splitting* to emulate polymorphism. Their analysis generally yields much more precise types than our soft typing framework, although the analysis is much more expensive in both time and space than our prototype. They use their analysis only to optimize run-time checks and do not attempt to produce useful type information for programmers. If useful type information could be extracted from their analysis, it could serve as a foundation for a more precise, albeit more expensive, soft type system.

6.3 Static Typing

Statically typed languages provide all the benefits of types and type checking that we seek, but at the price of rejecting programs that do not satisfy the type system's arbitrary constraints. Because type checking is a form of program verification, any static type checker will reject some programs that do not actually misinterpret data. These programs satisfy all the abstract criteria of type correctness and are often useful and concise. They are rejected merely because of the inherent incompleteness of the type checker. This compromises the ability of statically typed languages to address new paradigms such as object-oriented programming and nourishes a never-ending search for better static type systems. Consequently, realistic statically typed languages such as C usually have some means of circumventing the type checker. These type loopholes make debugging and writing portable code more difficult. Modern type-safe languages such as Standard ML [Milner et al. 1990] and Modula 3 [Cardelli et al. 1989] have sophisticated type systems that include polymorphism and subtyping to achieve greater flexibility, but even implementations of these languages invariably include loopholes, such as the procedure `System.Unsafe.cast` in Standard ML of New Jersey.

Several authors [Abadi et al. 1991; 1995; Leroy and Mauny 1993] have developed extensions that add a *dynamic* type to the ML type system. These extensions provide an explicit operation that pairs a statically typed value with its type to yield a self-describing value of type *dynamic*. Values of type *dynamic* can be passed to and returned from procedures and even exported beyond the programming language environment. A value of type *dynamic* can only be used by projecting it back to its static type, which requires a run-time check. This projection operation is usually incorporated in the programming language by extending the pattern-matching construct. Because of their explicit nature, *dynamic* type extensions are primarily useful for limited applications such as persistent storage. These extensions do not alter the fundamental character of static type systems.

Freeman and Pfenning [1991] and Freeman [1993] have developed a system of *refinement types* for ML that permits more precise type assignment within datatypes. Their system does not expand the set of typable programs. Rather, with the aid of explicit annotations, refinement types permit more precise static checking of the use of variants within datatypes. Refinement types enable the compile-time detection of more errors and the elimination of some compiler warnings and run-time checks for variants. But again, refinement types do not alter the fundamental character of static type systems.

7. CONCLUSION

Our prototype uses an internal representation for types that smoothly extends the Hindley-Milner type discipline with limited union types and recursive types. This representation is expressive, compact, efficient to compute, and supports the incremental definition of new type constructors. When viewed as a static type system, our internal type system satisfies the usual type soundness property (well-typed programs cannot go wrong). When used as a soft type system with an algorithm that inserts run-time checks, a correspondence property ensures that soft typed programs exhibit the same behavior as their dynamically typed counterparts.

The presentation types our system provides to programmers are natural and easy to interpret. While principal presentation types do not exist, the translation from internal types to presentation types ensures that the presentation type displayed for an expression approximates its internal type. That is, a presentation type may denote a larger subset of the data domain than its corresponding internal type, but the reverse cannot happen.

Our prototype accommodates the "grubby" features of realistic programming languages, such as uncurried procedures of fixed and variable arity, assignment, and first-class continuations. Support for pattern matching, data definition, and immutable data facilitates more precise type assignment. The prototype does not include support for separate compilation.

On a suite of benchmarks that range in size up to 6000 lines of Scheme code, our prototype typically eliminates 90% of the run-time checks that are necessary for safe execution without soft typing. Consequently, most soft typed programs run 10 to 15% faster than their dynamically typed counterparts, and a few run more than twice as fast. For programs of less than 1000 lines, type analysis usually takes less than a second. For our largest benchmark (the prototype itself), type analysis takes 18 seconds on a 120MHz Pentium.

For existing Scheme programs, Soft Scheme is valuable as an optimization tool to safely eliminate run-time checks. It is less useful as a diagnostic aid to programmers for two reasons. First, the types it assigns to expressions and identifiers are often too imprecise to be useful to the programmer. Second, our prototype does not include a suitable mechanism for explaining to the programmer why a particular run-time check has been inserted. But when programs are developed with soft typing in mind, our prototype can provide valuable guidance in structuring programs to avoid run-time checks.

Soft Scheme and the pattern matching and data definition extensions for Scheme are freely available from http://www.neci.nj.nec.com/homepages/wright.html.

APPENDIX

A. EXAMPLES

Following are some simple functions with their inferred types. None of these functions require run-time checks if they are passed appropriate arguments.

```
(define map     ; apply a function to every element of a list
  (λ (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
  (λ (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 nonempty list
  (λ (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
  (λ (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    ; concatenate argument lists
  (λ x
    (cond [(null? x) ()]
          [(null? (cdr x)) (car x)]
          [else (let loop ([m (car x)])
                  (if (null? m)
                      (apply append (cdr x))
                      (cons (car m) (loop (cdr m)))))])))
;; ((arglist (list X1)) ->* (list X1))
```

```
(define taut?     ; test for a tautology
  (λ (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 et al. [1994]
(define Y     ; least fixed point combinator
  (λ (f)
    (λ (y)
      (((λ (x) (f (λ (z) ((x x) z))))
        (λ (x) (f (λ (z) ((x x) z)))))
       y))))
;; (((X1 -> X2) -> (X1 -> X2)) -> (X1 -> X2))
(define last    ; find last element of a list
  (Y (λ (f)
       (λ (x)
         (if (null? (cdr x))
             (car x)
             (f (cdr x)))))))
;; ((cons Z1 (list Z1)) -> Z1)
```

B. SEMANTICS OF TYPES

We used operational semantics throughout this article because we believe that language semantics and proofs of type soundness are often expressed most clearly in an operational framework [Wright and Felleisen 1994]. But denotational models of types as sets of values can also lend valuable intuition to reasoning about programs and types. In this appendix, we present two denotational models of types as sets of values for Core Scheme. We present both a denotational model of internal types and presentation types, and we show that the denotation of an internal type is always contained in the denotation of its presentation form.

B.1 Internal Types

We use the ideal model developed by MacQueen et al. [1986] to assign meaning to the internal types of Section 2.3.

The value domain is the solution of the usual reflexive domain equation

$$\mathbf{D} = \mathbf{N}_\perp \oplus \mathbf{T}_\perp \oplus \mathbf{F}_\perp \oplus \mathbf{I}_\perp \oplus (\mathbf{D} \otimes \mathbf{D}) \oplus [\mathbf{D} \to_{sc} \mathbf{D}]_\perp \oplus \mathbf{E}_\perp \oplus \mathbf{W}_\perp$$

where $\oplus$ is the coalesced tagged sum construction on domains; $\otimes$ is the strict Cartesian product construction on domains; $\to_{sc}$ is the strict continuous function space construction on domains; $[\,]_\perp$ is the lifting construction on domains; $\mathbf{N}_\perp$ is the flat domain of numbers; $\mathbf{T}_\perp$ is the domain $\{\#\mathsf{t}\}_\perp$; $\mathbf{F}_\perp$ is the domain $\{\#\mathsf{f}\}_\perp$; $\mathbf{I}_\perp$ is the domain $\{\text{'}()\}_\perp$; $\mathbf{E}_\perp$ is the domain $\{\mathsf{check}\}_\perp$; and $\mathbf{W}_\perp$ is the domain $\{\mathsf{wrong}\}_\perp$. Other flat domains can be included as desired. $\mathbf{E}_\perp$ introduces an error element that is returned by invalid applications of checked operations. Our semantics includes check in every type. $\mathbf{W}_\perp$ introduces an error element for invalid operations of unchecked operations. The error element wrong is not a member of any type. The product ($\otimes$) and function space ($\to_{sc}$) constructors are strict in both check and wrong, as well as $\perp$.

The meaning of a type τ is an *ideal* over $\mathbf{D}$. An ideal is a set of values that is closed downward under approximations, and closed upward to least upper bounds of consistent sets of values. Let $\mathcal{I}(\mathbf{D})$ be the set of ideals of $\mathbf{D}$. To assign an ideal to open types, we need an interpretation for the type's free variables. A function $\rho \in \mathit{TypeEnv}$ maps the free type variables of a type to certain ideals of $\mathbf{D}$ and the free flag variables to the set $\{\mathbf{+},\mathbf{-}\}$:

$$\begin{array}{rcl} \mathit{TypeEnv} &=& \mathit{TypeVar}^{\{\kappa_1,\ldots,\kappa_n\}} \to \mathcal{I}(\mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}}) \\ && \cup\ \mathit{FlagVar} \to \{\mathbf{+},\mathbf{-}\}. \end{array}$$

The labels $\{\kappa_1, \ldots, \kappa_n\}$ on type variables that enforce tidiness restrict the sets of ideals that type variables may denote. Let $\mathbf{D}_\kappa$ name that subset of $\mathbf{D}$ that corresponds to type constructor κ, i.e., $\mathbf{D}_{num} = \mathbf{N}_\perp$, $\mathbf{D}_{cons} = \mathbf{D} \otimes \mathbf{D}$, $\mathbf{D}_{\to} = [\mathbf{D} \to_{sc} \mathbf{D}]_\perp$, etc. $\mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}}$ is that subset of $\mathbf{D}$ that excludes elements from $\mathbf{D}_{\kappa_1}$ through $\mathbf{D}_{\kappa_n}$ other than check and $\perp$. That is,

$$\mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}} \;=\; (\mathbf{D} - \bigcup_{\kappa\in\{\kappa_1,\ldots,\kappa_n\}} \mathbf{D}_\kappa) \cup \mathbf{E}_\perp.$$

For notational convenience, let $\rho(\mathbf{+}) = \mathbf{+}$ and $\rho(\mathbf{-}) = \mathbf{-}$ for any type environment ρ.

Given an assignment for the free variables of a type, the semantic function $\mathcal{T}$: $\mathit{Type} \to \mathit{TypeEnv} \to \mathcal{I}(\mathbf{D})$ defined in Figure 14 maps a type to an ideal of $\mathbf{D}$. We define the product ($\boxtimes$) and exponentiation ($\boxed{\to}$) functions on ideals as follows:

$$\begin{array}{rcl} I \boxtimes J &=& I \otimes J \\ I \boxed{\to} J &=& \{f \in \mathbf{D} \to_{sc} \mathbf{D} \mid f(I) \subseteq J\} \end{array}$$

and we identify their results with the corresponding subsets of $\mathbf{D}$.[16] Where f :

[16]Technically, $I \boxtimes J$ and $I \boxed{\to} J$ are only *isomorphic* to corresponding subsets of $\mathbf{D}$; see MacQueen et al. [1986, p. 104].

$$
\begin{array}{rcl}
\mathcal{T}[\![\not\emptyset]\!]\rho & = & \mathbf{E}_\bot \\
\mathcal{T}[\![num^f \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathbf{N}_\bot \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![true^f \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathbf{T}_\bot \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![false^f \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathbf{F}_\bot \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![nil^f \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathbf{I}_\bot \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![(cons^f\ \sigma_1\ \sigma_2) \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathcal{T}[\![\sigma_1]\!]\rho \boxtimes \mathcal{T}[\![\sigma_2]\!]\rho \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![(\sigma_1 \rightarrow^f \sigma_2) \cup \tau]\!]\rho & = & (\text{if } \rho(f) = \text{+} \text{ then } \mathcal{T}[\![\sigma_1]\!]\rho \boxed{\rightarrow} \mathcal{T}[\![\sigma_2]\!]\rho \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![\alpha]\!]\rho & = & \rho(\alpha) \\
\mathcal{T}[\![\mu\alpha.\tau]\!]\rho & = & \mu(\lambda I \in \mathcal{I}(\mathbf{D}).\ \mathcal{T}[\![\tau]\!](\rho[\alpha \mapsto I])) \\
\mathcal{T}[\![\forall\alpha^{\{\kappa_1,\ldots,\kappa_n\}}.\tau]\!]\rho & = & \forall_{\mathcal{U}}(\lambda I \in \mathcal{I}(\mathbf{D}).\ \mathcal{T}[\![\tau]\!](\rho[\alpha \mapsto I])) \text{ where } \mathcal{U} = \mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}} \\
\mathcal{T}[\![\forall\varphi.\tau]\!]\rho & = & \mathcal{T}[\![\tau]\!](\rho[\varphi \mapsto \text{+}]) \cap \mathcal{T}[\![\tau]\!](\rho[\varphi \mapsto \text{-}])
\end{array}
$$

Fig. 14. Ideal semantics for internal types.

$\mathcal{I}(\mathbf{D}) \rightarrow \mathcal{I}(\mathbf{D})$ is a function of one argument, let

$$\forall_{\mathcal{U}} f \;=\; \bigcap_{I \in \mathcal{U}} f(I).$$

For an appropriate function $f : \mathcal{I}(\mathbf{D}) \rightarrow \mathcal{I}(\mathbf{D})$, let μf be the unique least element $x \in \mathcal{I}(\mathbf{D})$ such that $x = f(x)$. As all internal type expressions are formally contractive, all types have an interpretation.

B.2 Presentation Types

We use the same model to assign meaning to the presentation types of Section 3.

Because presentation types do not include flags, the environments that provide values for open types need only map type variables to ideals of $\mathbf{D}$:

$$\mathit{TypeEnv} \;=\; \mathit{TypeVar}^{\{\kappa_1,\ldots,\kappa_n\}} \rightarrow \mathcal{I}(\mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}})$$

The superscripts $\{\kappa_1, \ldots, \kappa_n\}$ on presentation type variables are implicit. The position of a type variable in a type determines this label, and a type is well formed only if all its type variables have uniquely determined labels.

Given an assignment for the free variables of a type, the semantic function $\mathcal{P}$: $\mathit{Type} \rightarrow \mathit{TypeEnv} \rightarrow \mathcal{I}(\mathbf{D})$ defined in Figure 15 maps a presentation type to an ideal of $\mathbf{D}$. The extra occurrences of $\mathbf{E}_\bot$ that do not arise in the semantics for internal types ensure that check is a member of singular union types such as num. In the semantics for internal types, the corresponding type $num^{\text{+}} \cup \not\emptyset$ includes check because $\not\emptyset$ includes check.

The semantics treats a simpler form of recursive type $\mu X.T$, rather than the first-order recurrence equations of presentation types. The following function translates recursive presentation types to this intermediate form:

$$
\begin{array}{l}
\mathcal{R}[\![(rec\ ()\ T)]\!] \;=\; T \\
\mathcal{R}[\![(rec\ ([X_1\ T_1])\ T_0)]\!] = \; T_0[X_1 \mapsto \mu X_1.T_1] \\
\mathcal{R}[\![(rec\ ([X_1\ T_1][X_2\ T_2]\ldots)\ T_0)]\!] \\
\quad = \; \mathcal{R}[\![(rec\ ([X_2\ T_2]\ldots)\ T_0)]\!][X_1 \mapsto \mu X_1.\mathcal{R}[\![(rec\ ([X_2\ T_2]\ldots)\ T_1)]\!]]
\end{array}
$$

Again, as all presentation type expressions are formally contractive, all types have an interpretation.

$$
\begin{aligned}
\mathcal{P}[\![\mathit{num}]\!]\rho &= \mathbf{N}_\perp \cup \mathbf{E}_\perp \\
\mathcal{P}[\![\mathit{true}]\!]\rho &= \mathbf{T}_\perp \cup \mathbf{E}_\perp \\
\mathcal{P}[\![\mathit{false}]\!]\rho &= \mathbf{F}_\perp \cup \mathbf{E}_\perp \\
\mathcal{P}[\![\mathit{nil}]\!]\rho &= \mathbf{I}_\perp \cup \mathbf{E}_\perp \\
\mathcal{P}[\![(\mathit{cons}\ T_1\ T_2)]\!]\rho &= (\mathcal{P}[\![T_1]\!]\rho \boxtimes \mathcal{P}[\![T_2]\!]\rho) \cup \mathbf{E}_\perp \\
\mathcal{P}[\![(T_1 \ \texttt{->}\ T_2)]\!]\rho &= (\mathcal{P}[\![T_1]\!]\rho \boxed{\to} \mathcal{P}[\![T_2]\!]\rho) \cup \mathbf{E}_\perp \\
\mathcal{P}[\![X]\!]\rho &= \rho(X) \\
\mathcal{P}[\![(+\ P_1 \ldots P_n)]\!]\rho &= \mathcal{P}[\![P_1]\!]\rho \cup \ldots \cup \mathcal{P}[\![P_n]\!]\rho \cup \mathbf{E}_\perp \\
\mathcal{P}[\![(+\ P_1 \ldots P_n\ N_1 \ldots N_m\ X)]\!]\rho &= \mathcal{P}[\![P_1]\!]\rho \cup \ldots \cup \mathcal{P}[\![P_n]\!]\rho \cup \mathcal{P}[\![X]\!]\rho \\
\mathcal{P}[\![\mu X.T]\!]\rho &= \mu(\lambda I \in \mathcal{I}(\mathbf{D}).\ \mathcal{P}[\![T]\!](\rho[X \mapsto I])) \\
\mathcal{P}[\![\forall X^{\{\kappa_1,\ldots,\kappa_n\}}.T]\!]\rho &= \forall_{\mathcal{U}}(\lambda I \in \mathcal{I}(\mathbf{D}).\ \mathcal{P}[\![T]\!](\rho[X \mapsto I])) \text{ where } \mathcal{U} = \mathbf{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}}
\end{aligned}
$$

Fig. 15. Ideal semantics for presentation types.

B.3 Translating to Presentation Types

Section 3 defines a translation from internal types to presentation types. We show that the denotation of an internal type is a subset of the denotation of its presentation type.

Lemma B.3.1. *If T is the presentation form of the closed internal type τ, then $\mathcal{T}[\![\tau]\!]\emptyset \subseteq \mathcal{P}[\![T]\!]\emptyset$.*

Proof. Translating an internal type τ to a presentation type T involves three steps.

(1) Replace all useless flag variables in τ with $\mathbf{-}$ and all useless type variables in τ with $\varnothing$. Let the result of this step be $\tau' = \mathit{useless}(\tau)$.
(2) Replace all other flag variables with $\mathbf{+}$. Let the result of this step be $\tau'' = \mathit{unflag}(\tau')$.
(3) Translate the type to presentation type syntax. The result of this step is $T = \mathit{to\text{-}presentation}(\tau'')$.

We need to show that the first two steps preserve all elements of type τ. That is, we must show that

$$\mathcal{T}[\![\tau]\!]\emptyset \subseteq \mathcal{T}[\![\mathit{useless}(\tau)]\!]\emptyset \subseteq \mathcal{T}[\![\mathit{unflag}(\mathit{useless}(\tau))]\!]\emptyset.$$

The first step, *useless*, replaces flag (respectively, type) variables that occur only positively in τ with $\mathbf{-}$ (respectively, $\varnothing$). We show by induction on the depth of nesting within the first argument of function constructors that $\mathcal{T}[\![\tau]\!]\emptyset = \mathcal{T}[\![\mathit{useless}(\tau)]\!]\emptyset$. The induction depends on the contravariance property of the function space.

The second step, *unflag*, changes any remaining quantified flag variables to $\mathbf{+}$. That is, $\mathit{unflag}(\forall\varphi.\tau) = \tau[\varphi \mapsto \mathbf{+}]$. We show that $\mathcal{T}[\![\forall\varphi.\tau]\!]\emptyset \subseteq \mathcal{T}[\![\tau[\varphi \mapsto \mathbf{+}]]\!]\emptyset$ as follows:

$$
\begin{aligned}
\mathcal{T}[\![\forall\varphi.\tau]\!]\emptyset &= \mathcal{T}[\![\tau]\!]\emptyset[\varphi \mapsto \mathbf{+}] \cap \mathcal{T}[\![\tau]\!]\emptyset[\varphi \mapsto \mathbf{-}] && \text{by the definition of } \mathcal{T}, \\
&\subseteq \mathcal{T}[\![\tau]\!]\emptyset[\varphi \mapsto \mathbf{+}] && \text{by the definition of set intersection,} \\
&= \mathcal{T}[\![\tau[\varphi \mapsto \mathbf{+}]]\!]\emptyset && \text{by a simple substitution lemma.}
\end{aligned}
$$

This completes the proof. □

ACKNOWLEDGMENTS

Kent Dybvig extended Chez Scheme 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. Suresh Jagannathan wrote the second version of Nucleic. Shriram Krishnamurthi exercised Soft Scheme to develop the second, faster version of Slatex. Matthias Felleisen provided invaluable advice, direction, and criticism throughout this work. And finally, comments from the anonymous referees helped to clarify the presentation.

REFERENCES

Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. 1991. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst. 13,* 2 (Apr.), 237–268.

Abadi, M., Cardelli, L., Pierce, B., and Rémy, D. 1995. Dynamic typing in polymorphic languages. *J. Funct. Program. 5,* 1 (Jan.), 111–130.

Aiken, A. and Fahndrich, M. 1995. Dynamic typing and subtype inference. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture.* ACM Press, New York, 182–191.

Aiken, A. and Wimmers, E. L. 1993. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture.* ACM Press, New York, 31–41.

Aiken, A., Wimmers, E. L., and Lakshman, T. K. 1994. Soft typing with conditional types. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 163–173.

Amadio, R. M. and Cardelli, L. 1990. Subtyping recursive types. In *Proceedings of the 17th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 104–118.

Barendregt, H. P. 1984. *The Lambda Calculus: Its Syntax and Semantics.* Revised ed. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam.

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

Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. 1989. The Modula-3 type system. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, New York, 202–212.

Cartwright, R. and Fagan, M. 1991. Soft typing. In *Proceedings of the '91 Conference on Programming Language Design and Implementation.* ACM Press, New York, 278–292.

Cartwright, R. and Felleisen, M. 1994. Extensible denotational language specifications. In *Symposium on Theoretical Aspects of Computer Science.* Springer-Verlag, Berlin, 244–272.

Chambers, C. 1992. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. thesis, Stanford Univ., Stanford, Calif.

Clinger, W. and Rees, J. 1991. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers 4,* 3 (July), 1–55.

Damas, L. M. M. 1985. Type assignment in programming languages. Ph.D. thesis, Univ. of Edinburgh, Edinburgh, Scotland.

Dzeng, H. and Haynes, C. T. 1994. Type reconstruction for variable-arity procedures. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 239–249.

Fagan, M. 1990. Soft typing: An approach to type checking for dynamically typed languages. Ph.D. thesis, Rice Univ., Houston, Tex.

Feeley, M., Turcotte, M., and Lapalme, G. 1994. Using MultiLisp for solving constraint satisfaction problems: An application to nucleic acid 3D structure determination. *J. Lisp Symbol. Comput. 7,* 2/3, 231–248.

Felleisen, M. 1991. On the expressive power of programming languages. *Sci. Comput. Program. 17,* 35–75.

Felleisen, M. and Hieb, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci. 102*, 235–271.

Flanagan, C. and Felleisen, M. 1995. Set-based analysis for full Scheme and its use in soft-typing. Tech. Rep. TR95-253, Rice Univ., Houston, Tex. Oct.

Freeman, T. 1993. Refinement types. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, Pa.

Freeman, T. and Pfenning, F. 1991. Refinement types for ML. In *Proceedings of the '91 Conference on Programming Language Design and Implementation.* ACM Press, New York, 268–277.

Gomard, C. K. 1990. Partial type inference for untyped functional programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming.* ACM Press, New York, 282–287.

Greengard, L. 1987. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, New York.

Greiner, J. 1993. Standard ML weak polymorphism can be sound. Tech. Rep. CMU-CS-93-160R, Carnegie Mellon Univ., Pittsburgh, Pa. Sept.

Harper, R. and Lillibridge, M. 1993. Explicit polymorphism and CPS conversion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, New York, 206–219.

Heintze, N. 1993. Set based analysis of ML programs. Tech. Rep. CMU-CS-93-193, Carnegie Mellon Univ., Pittsburgh, Pa. July.

Henglein, F. 1992a. Dynamic typing. In *Proceedings of the European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 233–253.

Henglein, F. 1992b. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 205–215.

Henglein, F. and Rehof, J. 1995. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture.* ACM Press, New York, 192–203.

Hindley, R. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*, 29–60.

Hindley, R. J. and Seldin, J. P. 1986. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, New York.

Hoang, M., Mitchell, J., and Viswanathan, R. 1993. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings of the 8th Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif., 15–25.

Hudak, P., Jones, S. P., Wadler, P., Boutel, B., Fairbarn, J., Fasel, J., Guzman, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. 1992. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Not. 27,* 5 (May), Section R.

Jagannathan, S. and Weeks, S. 1994. Analyzing stores and references in a parallel symbolic language. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 294–305.

Jagannathan, S. and Wright, A. K. 1995. Effective flow analysis for avoiding run-time checks. In *Proceedings of the 2nd International Static Analysis Symposium.* Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 207–224.

Kaes, S. 1992. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 193–204.

Kaplan, M. A. and Ullman, J. D. 1980. A scheme for the automatic inference of variable types. *J. ACM 27,* 1 (Jan.), 128–145.

Kind, A. and Friedrich, H. 1993. A practical approach to type inference in EuLisp. *J. Lisp Symbol. Comput. 6,* 1/2 (Aug.), 159–175.

Leroy, X. 1992a. Typage polymorphe d'un langage algorithmique. Ph.D. thesis, L'Université Paris 7, France.

LEROY, X. 1992b. Unboxed objects and polymorphic typing. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 177–188.

LEROY, X. AND MAUNY, M. 1993. Dynamics in ML. *J. Funct. Program. 3,* 4, 431–463.

LEROY, X. AND WEIS, P. 1991. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 291–302.

MA, K. L. AND KESSLER, R. R. 1990. TICL—A type inference system for Common Lisp. *Softw. Pract. Exper. 20,* 6 (June), 593–623.

MACLACHLAN, R. A. 1992. The Python compiler for CMU Common Lisp. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 235–246.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1986. An ideal model for recursive polymorphic types. *Inf. Contr. 71*, 95–130.

MARTELLI, A. AND MONTANARI, U. 1976. Unification in linear time and space: A structured presentation. Tech. Rep. B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy. July.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17,* 348–375.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

MITCHELL, J. C. 1991. Type inference with simple subtypes. *J. Funct. Program. 1,* 3 (July), 245–286.

O'KEEFE, P. M. AND WAND, M. 1992. Type inference for partial types is decidable. In *Proceedings of the European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 408–417.

PATERSON, M. S. AND WEGMAN, M. N. 1978. Linear unification. *J. Comput. Syst. Sci. 16,* 2 (Apr.), 158–167.

RÉMY, D. 1989. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 77–87.

RÉMY, D. 1991. Type inference for records in a natural extension of ML. Tech. Rep. 1431, INRIA, France. May.

RÉMY, D. 1992. Extension of ML type system with a sorted equational theory on types. Tech. Rep. 1766, INRIA, France. Oct.

SHAO, Z. AND APPEL, A. K. 1993. Smartest recompilation. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 439–450.

SHIVERS, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, Pa.

STEELE, G. L., JR. 1994. Building interpreters by composing monads. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 472–492.

TALPIN, J.-P. AND JOUVELOT, P. 1992. The type and effect discipline. In *Proceedings of the 7th Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif., 162–173.

THATTE, S. R. 1988. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium.* Lecture Notes in Computer Science, vol. 317. Springer-Verlag, Berlin, 615–629.

THATTE, S. R. 1990. Quasi-static typing. In *Proceedings of the 17th Annual Symposium on Principles of Programming Languages.* ACM Press, New York, 367–381.

TOFTE, M. 1990. Type inference for polymorphic references. *Inf. Comput. 89,* 1 (Nov.), 1–34.

WAND, M. 1987. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alamitos, Calif., 37–44.

WAND, M. 1991. Type inference for record concatenation and multiple inheritance. *Inf. Comput. 93*, 1–15.

WRIGHT, A. K. 1992. Typing references by effect inference. In *Proceedings of the 4th European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 473–491.

WRIGHT, A. K. 1994. Practical soft typing. Ph.D. thesis, Rice Univ., Houston, Tex.

WRIGHT, A. K. 1995. Simple imperative polymorphism. *J. Lisp Symbol. Comput. 8,* 4, 343–356.

WRIGHT, A. K. AND CARTWRIGHT, R. 1994. A practical soft type system for Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 250–262.

WRIGHT, A. K. AND DUBA, B. F. 1993. Pattern matching for Scheme. Rice Univ., Houston, Tex. Available from http://www.neci.nj.nec.com/homepages/wright.html.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1 (Nov.), 38–94.

ZHAO, F. 1987. An $O(N)$ algorithm for three-dimensional N-body simulations. M.S. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass.

Received February 1996; revised August 1996; accepted October 1996