RICE UNIVERSITY

Practical Soft Typing

by

Andrew K. Wright

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Robert S. Cartwright, Co-chairman
Professor of Computer Science

Matthias Felleisen, Co-chairman
Professor of Computer Science

Robert Bixby
Professor of Computational and Applied Mathematics

Bruce F. Duba
Research Associate of Computer Science

Houston, Texas

August, 1994

Abstract

Practical Soft Typing

by Andrew K. Wright

Soft typing is an approach to type checking for dynamically typed languages. Like a static type checker, a soft type checker infers syntactic types for identifiers and expressions. But rather than reject programs containing untypable fragments, a soft type checker inserts explicit run-time checks to ensure safe execution.

Soft typing was first introduced in an idealized form by Cartwright and Fagan. This thesis investigates the issues involved in designing a practical soft type system. A soft type system for a purely functional, call-by-value language is developed by extending the Hindley-Milner polymorphic type system with recursive types and limited forms of union types. The extension adapts Remy's encoding of record types with subtyping to union types. The encoding yields more compact types and permits more efficient type inference than Cartwright and Fagan's early technique. Correctness proofs are developed by employing a new syntactic approach to type soundness. As the type inference algorithm yields complex internal types that are difficult for programmers to understand, a more familiar language of presentation types is developed along with translations between internal and presentation types. To address realistic programming languages like Scheme, the soft type system is extended to incorporate assignment, continuations, pattern matching, data definition, records, modules, explicit type annotations, and macros. Imperative features like assignment and continuations are typed by a new, simple method of combining imperative features with Hindley-Milner polymorphism.

The thesis shows soft typing to be practical by illustrating a prototype soft type system for Scheme. Type information determined by the prototype is sufficiently precise to provide useful diagnostic aid to programmers and to effectively minimize run-time checking. The type checker typically eliminates 90% of the run-time checks that are necessary for safe execution with dynamic typing. This reduction in run-time checking leads to significant speedup for some bench marks. Through several examples, the thesis shows how prototypes, developed using a purely semantic understanding of types as sets of values, can be transformed into robust maintainable, and efficient programs by rewriting them to accommodate better syntactic type assignment.

Acknowledgments

I owe a great debt to my thesis advisors, Corky Cartwright and Matthias Felleisen. In my early years at Rice, they gave me a solid introduction to advanced programming languages and programming language semantics. From their individual perspectives, they instilled in me a taste for simplicity and elegance. They guided my research in a direction suitable to my tastes, though those tastes differ somewhat from their own. I am particularly grateful to Matthias for tolerating my interest in types. I believe his views on types are now more moderate, and I am flattered to think that I played a rôle in changing his opinions.

Many people in the Rice community contributed to a stimulating educational environment. Bruce Duba has been a great sounding board for even my craziest ideas. Amr Sabry and I had many fruitful discussions, and he has been an ideal office mate. Mike Fagan took the first plunge into soft typing and helped me understand his original ideas. Hans Boehm helped me get started in research at Rice. The other students and faculty members in the department contributed, if nothing else, by making Rice a fun place to be.

Last but not least, I thank my family and Shireen Dadmehr for their love, steady support and encouragement.

My dissertation work was financed in various parts by Rice University, the United States Army under a National Defense Science and Engineering Graduate Scholarship, the National Science Foundation (NSF grant CCR-9122518), the Texas Advanced Technology Program (Grant 003604-014), and the National Science and Engineering Research Council of Canada.

Dedication

To my Dad.

Contents

Figures

Chapter 1

Introduction

Computer programs are among the most complex creations of man. The complete specification of a typical program like an editor, typesetting system, compiler, or network protocol involves an enormous amount of detail. Advanced functional languages like Scheme and ML include powerful constructs to relieve the programmer of the burden of specifying certain low-level details. For instance, first-class procedures obviate the need to manipulate records describing states of subcomputations, and automatic storage management eliminates explicit allocation, formatting, and deallocation of memory. But the development of any serious software system is still a complex, time-consuming, and error-prone task.

To reason about program behavior, it is often essential to know which subsets of the data domain can appear at various textual points in a program. These subsets are known as *static types* or simply *types*. Types abstract the sets of values that may be bound to identifiers, the shapes of data structures, the permissible inputs and outputs of procedures, and the results of expressions. Type information provides the following benefits.

(i) Type consistency checking can prevent possible *misinterpretation* of data prior to program execution. A program misinterprets data when it performs a primitive operation with arguments for which the operation is not defined. For example, the following C program misinterprets the integer `10` as a pointer to a string when it calls `puts`.[1]

```
main() {
  puts( 10 );   /* puts writes a string to standard output */
}
```

[1]Most C compilers happily accept this obviously wrong program without complaint. The behavior of this program varies from one machine to another. Of greater concern are programs that misinterpret data in more subtle ways or only with particular inputs.

Type information can be used to flag some primitive operations that may misinterpret data as *type errors*. ANSI C compilers that perform argument type checking issue a warning about the above call to `puts`.

(*ii*) Types guide the design and organization of programs by providing discipline and structure. Programmer-supplied type definitions make data structures more evident. For instance, the following ML definition defines a representation for binary trees with `real` leaves.

```
datatype tree = Leaf of real
              | Node of tree * tree
```

And the types of identifiers and expressions provide mechanically verified documentation that aids program maintenance. For instance, an ML procedure that sums the leaves of `tree`s has type `tree -> real`.

(*iii*) Types provide opportunities for compiler optimizations. With type information available to the compiler, fewer primitive operations require run-time checks and fewer data values need to carry run-time type tags for primitive operations to test. For example, type information can reveal that x is a list of numbers in the following Scheme program.

```
(define sum-list
  (lambda (x)
    (if (null? x)
        0
        (+ (car x) (sum-list (cdr x))))))
(sum-list '(1 2 3))
```

Hence + does not need to perform run-time checks for non-numeric arguments, and car and cdr (the projection functions for pairs) do not need run-time checks. Furthermore, since x consists only of numbers, the numbers in the list '(1 2 3) do not need tags to distinguish them from non-numbers.

This thesis explores the issues involved in designing a practical *soft type system* to provide type information for languages like Scheme. Soft typing [10, 16] is a generalization of traditional *static* typing that accommodates both *static* and *dynamic* typing in one framework. To introduce and motivate soft typing, we begin with a discussion of traditional static and dynamic type systems.

1.1 Static Type Systems

Static type systems like that of core ML partition the data domain into disjoint types called *datatypes*. Figure 1.1 indicates a typical partitioning of the data domain. In this

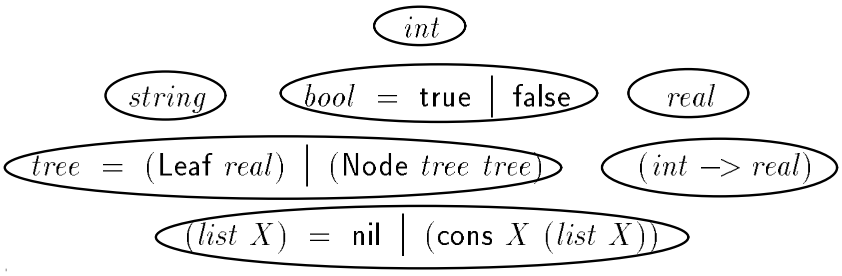

Figure 1.1 Datatypes for a Static Type System

example, *int*, *string*, and *real* are primitive datatypes; *(int –> real)* is the datatype that includes procedures taking an argument of type *int* and returning a result of type *real*; *bool* and *tree* are defined datatypes; and *(list X)* is a polymorphic datatype (X is a *type variable* that can stand for any type). Defined datatypes like *bool*, *tree*, and *(list X)* consist of values built from a finite collection of data constants and constructors that we collectively call *variants*. For instance, datatype *bool* consists of the constants true and false. A value of datatype *tree* is either a (Leaf $\cdot$) that is constructed from one *real* value, or a (Node $\cdot\ \cdot$) that is constructed from two *trees*.

A static type system assigns a datatype to each identifier and expression in a program. Static type systems prevent inadvertent mixing of values from different datatypes by rejecting programs that cannot be assigned datatypes in a consistent manner. Within a datatype, static type systems prevent confusion of variants by adding run-time tags to values of different variants. Primitive operations that accept only a subset of a datatype's variants test the tags of their arguments before performing their function. For example, the primitive that extracts the first element of a *(list X)* requires its argument be a cons variant.

By rejecting programs that mix values of different datatypes, static type systems detect some ways in which data may be misinterpreted prior to program execution. By assigning datatypes to identifiers and expressions, static type systems provide discipline and structure to guide the design and organization of programs. And since values of different datatypes cannot be mixed, static typing provides opportunities for compiler optimizations. Elements of different datatypes can have the same machine representation. Primitive operations do not need to test for values outside their argument's datatypes.

Although static typing in languages like ML yields the benefits of type information, it limits the flexibility of a programming language. Mixing values of different datatypes requires constructing additional data structures to inject values to be mixed into a common datatype. For example, a procedure argument that is conceptually either a *real* or a *string* must be passed as a *real-or-string*:

$$\textit{real-or-string} \;=\; (\mathsf{R}\ \textit{real}) \;\Big|\; (\mathsf{S}\ \textit{string}).$$

A passed argument of type *real-or-string* must be explicitly constructed using an R- or S-constructor. The procedure itself must explicitly project an input of type *real-or-string* to *real* or *string*. These explicit injections and projections add semantic complexity to programs as well as run-time overhead.

For these reasons, most widely used programming languages like C and Pascal incorporate only a weak form of static typing. Weak static type systems omit many of the run-time checks for variants within datatypes, and sometimes also allow datatypes to be mixed inappropriately. These weaker type systems undermine the benefits of static typing. No longer can the programmer be sure that an identifier is bound only to values belonging to its type. Nor is there any guarantee that a "typable" program will not misinterpret data. Debugging programs written in languages with weak static type systems can be extremely difficult because the actions of programs that misinterpret data cannot be understood from the semantics of the programming language alone. Understanding the behavior of a program that misinterprets data requires understanding a specific host machine's behavior.

1.2 Dynamic Type Systems

Dynamic type systems like that of Scheme ensure that programs cannot misinterpret data without limiting the flexibility of the programming language. Dynamic type systems do not partition the data domain into datatypes. Instead, all values carry tags and most primitive operations perform run-time checks on their arguments. A dynamic type system can be viewed as a static type system with only one datatype

$$D \ = \ \mathsf{num} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \mid (\mathsf{cons}\ D\ D) \mid (D \text{ -> } D)$$

Figure 1.2 Datatype for a Dynamic Type System

D, as in Figure 1.2. All values are variants within D. Hence all values have tags and procedures that do not accept all elements of D perform run-time checks.

Because dynamically typed languages have only one datatype, the type assignment methods used by static type systems are useless. If applied, these methods always yield a trivial consistent assignment of D to every expression and identifier. Thus dynamic type systems, while they ensure that programs cannot misinterpret data, are unable to provide the benefits of type information.

1.3 Soft Type Systems

Soft type systems adapt type inference techniques from static type systems to determine type information *within* a single (global) datatype D, as Figure 1.3 illustrates. A soft type system assigns subsets of D to identifiers and expressions. We refer to these subsets as *types*. Types represent upper bounds on the sets of values that can

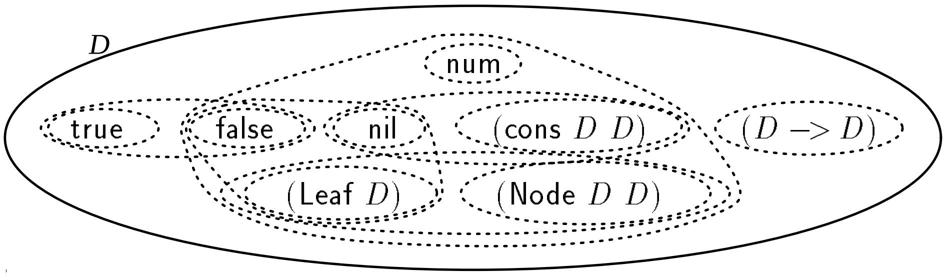

Figure 1.3 Types for a Soft Type System

be bound to identifiers and returned by expressions. A soft type system provides these types to the programmer to facilitate structuring, documenting, and debugging programs. In compiled programs, not all primitive operations incorporate run-time checks. Instead, a soft type system inserts run-time checks where the inferred type information indicates that primitive operations may be applied to invalid arguments. Thus a soft type system infers type information for the benefit of

- *the programmer*, to help structure, document, and debug programs; and
- *the compiler*, to produce more efficient code containing fewer run-time checks.

Since soft type systems do not partition the data domain into datatypes, they do not reject programs. Soft type systems provide the full flexibility of dynamically typed languages. Any program that can be executed with a dynamic type system can also be executed with a soft type system. By identifying primitive operations that may be applied to invalid arguments, soft typing helps avoid errors that may result in misinterpretation of data. By inferring type information for programs, soft type systems guide the design and organization of programs almost as much as static type systems do. Finally, the type information inferred by a soft type system provides opportunities for compiler optimizations. In particular, soft type systems optimize the placement of run-time checks and the representations of data.

1.4 Practical Soft Typing

Cartwright and Fagan [10, 16] pioneered soft typing in 1991. Their work, which we discuss in greater detail later, established the theoretical feasibility of soft typing for a idealized core language. They found a type inference algorithm that assigns reasonably precise types to all purely functional programs. They also developed an algorithm to insert run-time checks based on the inferred type information. But their work left many questions about soft typing unanswered.

(*i*) Can a soft type system provide type information that is useful to programmers?

(*ii*) How should soft type information and untypable program fragments be reported to the programmer?

(*iii*) Can soft type inference be fast enough to provide reasonable response?

(*iv*) Can soft type inference be precise enough to eliminate the majority of unnecessary run-time checks?

(*v*) How significant will the elimination of run-time checks be to program execution time?

In short, is soft typing practical?

This dissertation establishes the following thesis.

> **Soft typing is a practical approach to determining type information for dynamically typed languages.**

To support this thesis, we develop a new and comprehensive soft type theory. Our theory assigns concise and reasonably precise types to programs. The theory accommodates several features of practical programming languages, like procedures of varying arity, assignment, first-class continuations, user-defined data, and pattern matching. We develop correctness proofs in a formal setting for our system, and discuss the implementation of an efficient type inference algorithm. Finally, we describe an implementation of a soft type system for R4RS Scheme [13] and we discuss the effectiveness of our prototype from the perspectives of the programmer and of the compiler. From the programmer's perspective, our system helps structure programs and find bugs. From the compiler's perspective, our system eliminates many run-time checks from realistic programs and can significantly reduce program execution times.

Through experiments with our prototype, we have developed a philosophy for programming with a soft typed language. We believe that soft typed languages facilitate both the rapid development of software prototypes and the evolution of these software prototypes into robust, efficient, and maintainable production programs. Dynamically typed languages like Lisp, Scheme, and Smalltalk have long been recognized as valuable tools for rapid software prototyping. Statically typed languages like ML and Ada are recognized to provide valuable structure that increases the efficiency, reliability, and maintainability of programs. Because all dynamically typed programs can be executed with a soft type system, prototyping can be done the same way in a soft typed language as in a dynamically typed language. Programmers can reason about programs using a purely semantic understanding of a program, without regard to syntactic type assignment. Then as modules of a prototype stabilize, these modules can be rewritten to have better syntactic assignment. Just as with statically typed languages, better syntactic type assignment provides documentation for program maintenance and permits more efficient execution.

1.5 Thesis Outline

The next chapter introduces the notions of *dynamic typing*, *static typing*, and *soft typing* by illustrating them with a family of simple programming languages. We also introduce the notion of *type safety* and the formal techniques that we use throughout the remainder of the thesis for specifying language semantics and type systems.

The remainder of this thesis is organized into two parts. The first part, consisting of chapters 3 through 5, develops a soft type theory for realistic programming languages. Chapter 3 presents a formal definition of a soft type system for an idealized functional language called **Pure Scheme**. We develop this soft type system by enhancing a static type system. We adapt conventional techniques for proving type soundness for static type systems to proving the correctness of the soft type system. In Chapter 5, we extend our soft type system for **Pure Scheme** to encompass features of realistic programming languages like assignment, continuations, pattern matching, data definition, records, modules, and type annotations. Chapter 6 discusses work related to soft typing in general. We also comment on work related to specific aspects of soft typing throughout the thesis.

The second part of the thesis applies our soft type theory to construct a practical soft type system. Chapter 7 discusses our prototype soft type system for Scheme. We show how soft typing compares in practice to static and dynamic typing. We illustrate its effectiveness at optimizing run-time checks with some bench marks. And we show how our prototype may be used to engineer prototypes into robust and efficient programs. Chapter 8 discusses some problems with our work and suggests directions for future research.

This thesis includes three appendices. The first appendix presents proofs of several lemmas important to the correctness theorems in Chapter 3. The second appendix presents a denotational semantics of types as ideals for both our internal types and our presentation types. This semantics shows that presentation types are a conservative approximation to internal types. The third appendix is a user reference manual for our prototype soft type system, Soft Scheme.

Chapter 2

Dynamic, Static, and Soft Typing

In this chapter, we clarify the notions of *dynamic typing*, *static typing*, and *soft typing* by illustrating them with a family of simple programming languages. In each case we show how a *type safe* language prevents misinterpretation of data. Lexically-scoped, call-by-value, first-class procedures are a central feature of the programming languages we use. This family of languages forms the basis for modern programming languages like Scheme and core ML.

Throughout this chapter we also introduce our techniques for specifying language semantics and type systems. Specifically, we use *reduction semantics* to specify the meanings of programs and syntax-oriented deductive proof systems to define type systems. We also introduce the *Hindley-Milner static type discipline*, which underlies our soft type system.

2.1 Λ

We define a simple call-by-value functional language Λ that is based on the call-by-value λ-calculus [48]. Λ omits many features of realistic programming languages like interactive input/output and assignment, but it serves to illustrate the notions of *dynamic typing*, *static typing*, and *soft typing*.

Let *Id* be a denumerable set of *identifiers*[1] and *Const* be a set of *constants*. Λ has *expressions* (e) and *values* (v) of the forms

$$\begin{array}{rcl} e & ::= & v \;\Big|\; (e_1\ e_2) \\ v & ::= & c \;\Big|\; x \;\Big|\; (\mathbf{lambda}\ (x)\ e) \end{array}$$

where $x \in Id$ and $c \in Const$. The set of constants includes both *basic constants* like numbers and *primitive operations* like $+$, $-$, $*$, $\div$, etc. Λ is really a family of

[1] We reserve the term "variable" to refer to entities of type systems. We use "identifier" to refer to names in programs.

languages since we specify only some general properties that the set of constants must satisfy, rather than designate a specific set of constants. Values include constants, identifiers, and **lambda**-expressions that construct unnamed, first-class call-by-value procedures. Expressions include values and applications $(e_1\ e_2)$ of a procedure e_1 to a single argument e_2. The free and bound identifiers of an expression are defined as usual, with **lambda**-expressions binding their identifiers. Following Barendregt [7], 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. *Closed* expressions (resp. values) are expressions (resp. values) that have no free identifiers. *ClosedVal* is the set of closed values. *Programs* are closed expressions.

Since the primitive operations and **lambda**-expressions of Λ take only one argument, procedures requiring multiple arguments must be *curried.* A curried procedure is a higher-order procedure that takes the first argument and returns either an answer or a curried procedure that accepts the remaining arguments. Thus the binary function $+$ is a primitive that, when applied to the number n, returns a function $+n$ that, when applied to m, returns $n+m$. For the remainder of this chapter, we imagine Λ and its variants as having procedures accepting multiple arguments.

We specify the meaning of programs in Λ with a specific kind of term rewriting system called a *reduction semantics* [18], which is a form of operational semantics. The rewriting relation $\longmapsto$, defined in Figure 2.1, is a partial function mapping programs to programs. The definition is structured so that any closed expression other than a value can be uniquely decomposed into an *evaluation context* and a *redex.* An evaluation context E is an expression with a hole $[\,]$. The position of the hole indicates the next *redex*; that is, the next subexpression to be reduced by the rewriting relation. Redexes for Λ are of the form $((\textbf{lambda}\ (x)\ e)\ v)$ or $(c\ v)$. A redex of the form $((\textbf{lambda}\ (x)\ e)\ v)$ indicates application of a call-by-value procedure. The β_{v} rule reduces such a redex by substituting the argument value v for each occurrence of x in e, avoiding capture of free identifiers of v. We write $[x/v]e$ for this operation. A redex of the form $(c\ v)$ indicates the application of a primitive operation. A computable partial function[2]

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

[2]The notation $\rightharpoonup$ indicates a partial function.

$$
\begin{array}{rcll}
E[((\textbf{lambda}\ (x)\ e)\ v)] & \longmapsto & E[[x/v]e] & & (\beta_v) \\
E[(c\ v)] & \longmapsto & E[\delta(c,v)] & \text{if } \delta(c,v) \in \mathit{ClosedVal} & (\delta_1) \\
E[(c\ v)] & \longmapsto & \mathsf{check} & \text{if } \delta(c,v) = \mathsf{check} & (\delta_2)
\end{array}
$$

$$
E \ ::= \ [\,] \ \big| \ (E\ e) \ \big| \ (v\ E) \qquad (\mathit{Evaluation\ Context})
$$

Figure 2.1 Operational Semantics for Λ

interprets such applications. For a specific constant and argument value, δ may (i) yield a result value, (ii) yield the special token check, or (iii) be undefined. If δ yields a value, rule δ_1 places this value in the evaluation context to construct a new program. If δ yields check, rule δ_2 rewrites the program to check, which denotes an error message like "car is not a pair" or "division by zero". If δ is undefined, then $\longmapsto$ is undefined for this program.

Evaluation proceeds by repeatedly rewriting a program until it yields an answer. Answers are closed values or the special token check, which indicates an error:

$$
\mathit{Answers} = \mathit{ClosedVal} \cup \{\mathsf{check}\}.
$$

Not all programs evaluate to answers. Programs that do not terminate (*i.e.* "infinite loop") have an infinite reduction sequence. Programs that reach a state where $\longmapsto$ is undefined have no meaning. Hence an *evaluation function* based on the rewriting relation $\longmapsto$ is at best a partial function. The evaluation function for Λ is defined as

$$
\mathit{eval}(p) = \begin{cases} v & \text{if } p \longmapsto\!\!\!\!\!\rightarrow v \\ \mathsf{check} & \text{if } p \longmapsto\!\!\!\!\!\rightarrow \mathsf{check} \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

where $\longmapsto\!\!\!\!\!\rightarrow$ is the reflexive, transitive closure of $\longmapsto$ and p is a program.

Such an evaluation function may not completely reflect the behavior of an implementation of Λ. The behavior of a program executing on a computer cannot be "undefined". Given any program, the machine always does something. For a program that fails to terminate, the machine continues to compute without returning an answer. For a program that reaches a state where $\longmapsto$ is undefined, the machine will perform some implementation-dependent action and continue evaluation. Such an action may involve interpreting a number as a pointer, indexing beyond the end of an array, or otherwise misinterpreting data.

An implementation that continues to execute a non-terminating program forever raises no serious obstacle to developing robust, reliable programs—the implementation is being faithful to the semantics of the programming language. But an implementation that continues computing when the relation $\longmapsto$ is undefined can make debugging programs extremely difficult. The implementation-dependent actions taken after the "error" cannot be understood from the semantics of the programming language alone, and may not even be repeatable. These actions may eventually cause the implementation to generate a "core dump", but debugging from a core dump requires intimate knowledge of the underlying machine and of the language implementation. Without hardware memory protection, the machine-dependent actions taken after an error can cause a machine crash, leaving the programmer with no information as to the cause of the error. (MacIntosh users experience this problem on a daily basis.) Or the implementation may return a bogus "answer" that is indistinguishable from a valid answer returned by a correct program, giving the programmer no indication that an error occurred at all.

To separate the two ways in which *eval* may be undefined and to isolate programs that misinterpret data, we say that a program that reaches a state where $\longmapsto$ is undefined is *stuck*. For Λ, all stuck programs have the form

$E[(c\ v)]$ where $\delta(c, v)$ is undefined.

It is easy to show by induction on the length of the reduction sequence that all Λ-programs either

(i) yield an answer (that is either check or a closed value),

(ii) diverge (have an infinite reduction sequence), or

(iii) become stuck (reduce to a stuck program).

We say that a programming language that prevents programs from becoming stuck is *type safe*. For Λ, type safety requires that no primitive operation be performed with arguments for which it is not defined and no basic constant be applied. Both Scheme and ML are type safe, but they achieve type safety in different ways. Scheme uses dynamic typing, while ML uses static typing.

2.2 Dynamic Typing

Dynamically typed languages ensure type safety by performing run-time checks. To obtain a dynamically typed variant of Λ called $\Lambda_{\mathrm{DYNAMIC}}$, we simply insist that δ be a total function. When δ is total, there are no stuck programs—primitive operations cannot misinterpret data.

Since Λ-programs can only get stuck at constant applications, an implementation of $\Lambda_{\mathrm{DYNAMIC}}$ must guarantee that any constant can be applied to any value to yield either a result value or an error message. As constants are partitioned into basic constants and primitive operations, this means that (i) primitive operations may need to perform run-time checks on their arguments, and (ii) applications may need to perform run-time checks on their function values to ensure that the function subexpression of an application does not evaluate to a basic constant. When a run-time check fails, it aborts program execution with an error message. The semantics models failure of either kind of run-time check at an application $(c\ v)$ by defining $\delta(c, v) = \mathsf{check}$.

Type safe implementations of dynamically typed languages like $\Lambda_{\mathrm{DYNAMIC}}$ conceptually perform run-time checks at all primitive operations that accept a limited input domain. For example, the simple program

```
(let ([x (read)])
  (add1 x))
```

is implemented as

```
(let ([x (read)])
  (add1 (if (number? x)
            x
            (error "x is not a number")))).
```

The procedure **number?** is a predicate that tests whether its argument is a number, and **error** aborts program execution with an error message. To enable efficient implementation of a dynamically typed programming language, δ must be chosen such that the predicates of run-time checks can be computed efficiently. A common way to implement predicates is to designate a few bits (usually at least 2 and at most 16) of every value as a *tag*. Different tags indicate that the value is a number, pair, procedure, boolean, etc. Predicates simply test the tag value in constant time.

To achieve better performance, many implementations of dynamically typed languages attempt to eliminate some unnecessary run-time checks. But the extent of this optimization is usually limited, and the remaining run-time checks still add significant overhead to program execution. Hence many implementations allow the programmer to disable run-time type checking—omitting the run-time checks. In this mode, *valid* programs that do not trigger run-time check failures execute faster and give the same answers as they do under conventional "checked" execution. However, the language is no longer type safe. *Invalid* programs can produce arbitrary results ranging from "core dump" to erroneous but apparently valid answers.

2.3 Static Typing

A statically typed language achieves type safety by a combination of both run-time checks and constraints on programs. Not all stuck programs need be eliminated from the semantics by run-time checks. The constraints imposed by a static type system ensure that the remaining stuck programs cannot be reached from *typable* programs. Untypable programs that can lead to stuck programs are rejected by the type system and may not be executed.

A simple static type system for Λ_{STATIC}, a statically typed version of Λ, is based on the following set of *types*:

$$\tau \ ::= \ num \ \big| \ \tau_1 \rightarrow \tau_2.$$

Types denote sets of values. The type *num* denotes the set of all numbers. The type $\tau_1 \rightarrow \tau_2$ denotes the set of procedures that accept an argument of type τ_1 and return a result of type τ_2. A function *TypeOf* assigns a type to every constant in Λ_{STATIC}. With the set of types defined above, all constants must be either numbers or primitive operations.

A static type system for Λ_{STATIC} is a deductive proof system with *typing rules* that assign types to each of the expression forms. Figure 2.2 presents the axioms and inferences rules for Λ_{STATIC}. A type environment A is a finite map from identifiers to types. $A[x \mapsto \tau]$ denotes the functional extension or update of A at x to τ; $\emptyset$ denotes the empty finite map. 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 $\vdash e : \tau$, which implies that e is closed. A program is *typable* if a proof can be constructed according to this proof system. If a program is not typable, it cannot be executed. Figure 2.3 presents an example typing derivation for the program ((**lambda** (x) (add1 x)) 0).

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

$$\frac{\tau = A(x)}{A \vdash x : \tau} \qquad (\textbf{id})$$

$$\frac{A[x \mapsto \tau_1] \vdash e : \tau_2}{A \vdash (\textbf{lambda}\ (x)\ e) : \tau_1 \to \tau_2} \qquad (\textbf{lam})$$

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

Figure 2.2 Static Typing Rules for Λ_{STATIC}

$$\frac{\dfrac{\dfrac{\dfrac{\mathit{num} \to \mathit{num} = \mathit{TypeOf}(\mathsf{add1})}{A \vdash \mathsf{add1} : \mathit{num} \to \mathit{num}} \qquad \dfrac{\mathit{num} = A(\mathsf{x})}{A \vdash \mathsf{x} : \mathit{num}}}{A \vdash (\mathsf{add1}\ \mathsf{x}) : \mathit{num}}}{\emptyset \vdash (\textbf{lambda}\ (\mathsf{x})\ (\mathsf{add1}\ \mathsf{x})) : \mathit{num} \to \mathit{num}} \qquad \dfrac{\mathit{num} = \mathit{TypeOf}(\mathsf{0})}{\emptyset \vdash \mathsf{0} : \mathit{num}}}{\emptyset \vdash ((\textbf{lambda}\ (\mathsf{x})\ (\mathsf{add1}\ \mathsf{x}))\ \mathsf{0}) : \mathit{num}}$$

where $A = \emptyset[\mathsf{x} \mapsto \mathit{num}]$

Figure 2.3 Example Typing Derivation in Λ_{STATIC}

By rejecting untypable programs, Λ_{STATIC} prevents evaluation from reaching a stuck program. For example, the stuck program $(1\ 2)$ is untypable because 1 has type *num* but rule **ap** requires the first subexpression of a typable application to have a procedure type. It turns out that any program that evaluates to a stuck program is also untypable, as the proof of *Type Soundness* in Appendix A indicates. But to ensure that only untypable programs can reach stuck programs, some run-time checks are necessary. For instance, a program that divides a number by zero is typable in Λ_{STATIC}.[3] A run-time check for zero in $\div$ is necessary to abort execution. The following *typability* condition for Λ_{STATIC} indicates when run-time checks are required.

If $\mathit{TypeOf}(c) = \tau_1 \rightarrow \tau_2$ and $\vdash v : \tau_1$ then either
(i) $\delta(c, v) = v'$ and $\vdash v' : \tau_2$, or
(ii) $\delta(c, v) = \mathsf{check}$.

This condition requires that for a typable application $(c\ v)$, $\delta(c, v)$ must be defined and yield either a value of the result type of c or check. Given this typability condition, Λ_{STATIC} is type safe.

2.3.1 Datatypes

In Chapter 1, we described a static type system as partitioning the data domain into disjoint datatypes. The static type system presented above provides only the constant *num* and the constructor $\rightarrow$ from which to construct datatypes. Realistic programming demands the capability to define new datatypes.

Suppose we wish to define a new datatype to represent lists of numbers, as in

$$\mathit{list\text{-}of\text{-}num} \;=\; \mathsf{none} \;\Big|\; (\mathsf{one}\ \mathit{num}\ \mathit{list\text{-}of\text{-}num}).$$

Rather than fix a specific language mechanism for defining new datatypes, let us assume that the extension introduces constructors for building values of the new datatype and procedures for accessing their elements. Specifically, for our *list-of-num* datatype,

(i) none is an empty list constant;

[3] The constant 0 has the same type as every other number, hence the type system cannot distinguish division by zero from division by any other number.

(ii) one builds a new list by prefixing a number to an existing list;

(iii) first returns the first element of a list; and

(iv) rest returns a list consisting of all but the first element of a list.

In our semantics, the application of the constructor one to two values *is* a value, just as the constant none is a value. The behaviors of first and rest are specified by δ:

$$\begin{aligned}\delta(\mathsf{first}, (\mathsf{one}\ v_1\ v_2)) &= v_1 \\ \delta(\mathsf{rest}, (\mathsf{one}\ v_1\ v_2)) &= v_2.\end{aligned}$$

To extend the type system for Λ_{STATIC} to lists of numbers, we add types for the list operations as follows:

$$\begin{aligned}\mathsf{one} &: num \rightarrow (list\text{-}of\text{-}num \rightarrow list\text{-}of\text{-}num) \\ \mathsf{none} &: list\text{-}of\text{-}num \\ \mathsf{first} &: list\text{-}of\text{-}num \rightarrow num \\ \mathsf{rest} &: list\text{-}of\text{-}num \rightarrow list\text{-}of\text{-}num.\end{aligned}$$

Because first and rest take any argument of type $list\text{-}of\text{-}num$, they must accept the input none. Since the typability condition requires first and rest be defined for all arguments of their input type, we have

$$\begin{aligned}\delta(\mathsf{first}, \mathsf{none}) &= \mathsf{check} \qquad \text{and} \\ \delta(\mathsf{rest}, \mathsf{none}) &= \mathsf{check}.\end{aligned}$$

In other words, first and rest require run-time checks for none. Typability does not require that $\delta(\mathsf{first}, v)$ and $\delta(\mathsf{rest}, v)$ be defined for values v outside $list\text{-}of\text{-}num$. In other words, the type system guarantees that first and rest are never applied to values outside $list\text{-}of\text{-}num$.

Datatypes are an important tool for structuring programs. By introducing new datatype definitions, programmers can ensure that values of different datatypes are not inadvertently mixed. Primitive operations that accept all values of a datatype do not need to perform run-time checks. And an implementation need not represent values from different datatypes uniquely.

2.3.2 Polymorphism

While static typing is intended to aid the programming process, a static type system like the one introduced above can be as much hindrance as help. If the constraints imposed by the type system are too severe, programmers may have to expend considerable programming effort simply to accommodate the type system. For example, the above type system does not permit the definition of a list reversal procedure that reverses lists of elements of any type, even though the algorithm to reverse a list is independent of the types of the elements. Pascal is a prominent example of a programming language that suffers from this problem.

The Hindley-Milner type discipline [27, 43] extends the above static type system to permit the definition of *polymorphic* procedures. A procedure is polymorphic if its behavior does not depend on the type of some part of its arguments. For example, a list reversal procedure is polymorphic in the element type. A sorting procedure that takes a list of elements and a comparison function for elements is polymorphic in the element type.

To express the types of polymorphic procedures, we extend types with *type variables* (α) and introduce *type schemes* (Σ) as follows:

$$\begin{array}{lcl} \tau & ::= & num \;\Big|\; \tau_1 \rightarrow \tau_2 \;\Big|\; \alpha \\ \Sigma & ::= & \forall \vec{\alpha}.\, \tau \end{array}$$

(where $\vec{\alpha}$ stands for a sequence of zero or more α). Type variables stand for unknown types. The type scheme $\forall \vec{\alpha}.\, \tau$ binds type variables $\{\vec{\alpha}\}$ in τ. $FV(\Sigma)$ returns the free type variables of Σ. Type schemes represent the types of polymorphic procedures. For example, a polymorphic sorting procedure could have type scheme

$$\forall \alpha.\, (list\ \alpha) \rightarrow (\alpha \rightarrow \alpha \rightarrow bool) \rightarrow (list\ \alpha). \tag{2.1}$$

The first argument of type $(list\ \alpha)$ is the list of elements to be sorted; the second argument of type $(\alpha \rightarrow \alpha \rightarrow bool)$ is a comparison function for elements; and the sorting procedure returns a list of the same type as its first argument.

Type schemes describe sets of types by substitution for bound variables. A *substitution* S is a finite map from type variables to types. $S\tau$ means the simultaneous replacement of every free variable in type τ by its image under S. A type τ' is an *instance* of type scheme $\forall \vec{\alpha}.\, \tau$, written

$$\tau' \prec \forall \vec{\alpha}.\, \tau,$$

if there exists a substitution S with $\mathrm{Dom}(S) = \{\vec{\alpha}\}$ and $S\tau = \tau'$. For example, the type

$$(list\ num) \to (num \to num \to bool) \to (list\ num)$$

is an instance of type scheme 2.1 under the substitution $[\alpha/num]$.

Identifiers with polymorphic types are introduced by **let**-expressions. A **let**-expression is a new kind of expression that is not a value:

$$e \ ::= \ \ldots \ \big| \ (\mathbf{let}\ ([x\ e_1])\ e_2).$$

The above **let**-expression evaluates e_1 to a value, binds that value to x, and then evaluates e_2. We extend the semantics for Λ to include **let**-expressions by adding a reduction rule for **let**-expressions:

$$E[(\mathbf{let}\ ([x\ v])\ e)] \ \longmapsto \ E[[x/v]e]. \qquad (let)$$

We must also extend the set of evaluation contexts as follows:

$$E \ ::= \ \ldots \ \big| \ (\mathbf{let}\ ([x\ E])\ e).$$

The extended semantics reduces a **let**-expression by rule *let* once e_1 has been reduced to a value.

Polymorphic type schemes are assigned to identifiers bound by **let**-expressions. Where identifiers with a polymorphic type scheme are used, they may have any type that is an instance of their type scheme. To extend the type system of $\Lambda_{\mathrm{STATIC}}$ to perform polymorphic type assignment, we replace the typing rule for identifiers and add a typing rule for **let**-expressions. Figure 2.4 presents the new rules. The new rule **id** for identifiers generalizes the old rule. An identifier x may have any type that is an instance of its type scheme in type environment A. Rule **let** introduces polymorphic type schemes into type environments. Whenever the type τ_1 of a **let**-bound expression e_1 involves type variables, any of these type variables that can be freely renamed are quantified in the type scheme for x. Excluding type variables free in the type environment $(\ldots - FV(A))$ from quantification ensures that all quantified variables can be freely renamed.

To illustrate Hindley-Milner polymorphic typing, consider the expression

```
(let ([x (lambda (y) y)])
  (begin (x 1) (x none)))
```

$$\frac{\tau \prec A(x)}{A \vdash x : \tau} \qquad \textbf{(id)}$$

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

$$\mathit{Close}(\tau, A) \ \in\ \{\forall\vec{\alpha}.\,\tau \ \big|\ \{\vec{\alpha}\} \subseteq FV(\tau) - FV(A)\}$$

Figure 2.4 New Typing Rules for Polymorphism

where $(\textbf{begin}\ e_1\ e_2)$ abbreviates $((\textbf{lambda}\ (x)\ e_2)\ e_1)$ for $x \notin FV(e_2)$. A typing can be obtained for the subexpression (**lambda** (y) y) as follows:

$$\cfrac{\cfrac{\tau \prec (A[\mathsf{y} \mapsto \tau])(y)}{A[\mathsf{y} \mapsto \tau] \vdash \mathsf{y} : \tau}}{A \vdash (\textbf{lambda}\ (\mathsf{y})\ \mathsf{y}) : \tau \rightarrow \tau}$$

for any type τ. Hence we can choose $\tau = \alpha$ where α is a type variable that does not appear in $FV(A)$. As $\mathit{Close}(\alpha \rightarrow \alpha,\ A)$ includes $\forall\alpha.\,\alpha \rightarrow \alpha$, we can assign type scheme $\forall\alpha.\,\alpha \rightarrow \alpha$ to x. Where x is applied to the argument 1 of type *num*, x can be treated as an identity function for numbers because $(\mathit{num} \rightarrow \mathit{num}) \prec (\forall\alpha.\,\alpha \rightarrow \alpha)$ under the substitution $[\alpha/\mathit{num}]$. Where x is applied to none (the empty list of numbers), x can similarly be treated as an identity function for *list-of-num*.

2.4 Soft Typing

A soft typed language achieves type safety by performing run-time checks. But unlike a dynamically typed language that performs run-time checks at every primitive operation with a limited input domain, a soft type system first performs type analysis to determine types that represent upper bounds on the possible inputs to primitives. Based on this information, a soft type system inserts run-time checks at only those primitives whose input types are larger than their input domains. Unlike a static type system that places constraints on programs to achieve type safety, a soft type system enforces no constraints. All programs that can be executed under dynamic typing can also be executed with a soft type system.

The first task of a soft type system for Λ is to determine types for the identifiers and expressions of a program. The assigned types represent upper bounds (conservative approximations) on the sets of values that identifiers may take on and expressions may return. Type analysis suitable for a soft typed language Λ_{SOFT} must be able to cope with any Λ-program, hence the soft type system must include an upper bound for every set of values. The simplest soft type system for Λ simply adds a maximal type $\top$ to the types of $\Lambda_{\mathrm{STATIC}}$. The system assigns types to identifiers and expressions in a manner similar to $\Lambda_{\mathrm{STATIC}}$. But Λ_{SOFT} assigns $\top$ to an identifier or expression when it can assign no other type. For example, to type the following program:

```
(let ([x (read)])
  (let ([y (add1 x)])
    (sub1 y)))
```

Λ_{SOFT} assigns $\top$ to x since (read) can return any value. Identifier y has type *num* since y can only be bound to a number.

After assigning types, a soft type system inserts run-time checks by a transformation $e \Rightarrow e'$ from programs to programs. For Λ_{SOFT}, a primitive requires a run-time check if its input is $\top$. Λ_{SOFT} inserts run-time checks into the above program to yield

```
(let ([x (read)])
  (let ([y (add1 (if (number? x)
                     x
                     (error "x is not a number")))])
    (sub1 y))).
```

The system inserts a check around the input to add1 to ensure that the input is a number. The sub1 primitive requires no run-time check because the type of y indicates that sub1 is only ever applied to a number.

The run-time checks inserted by a soft type system should not alter the meaning of programs or their time or space complexity. That is, run-time checks should be simple functions that either fail or behave as the identity function. They should execute in constant time and constant space and they should be cheap so that a single ill-placed run-time check will not drastically affect a program's execution time. Operations like Scheme's list? that recursively descend through a value to determine whether it has a certain shape are not appropriate as run-time checks because their insertion can alter the complexity of a program. The same techniques used to implement run-time checks for dynamically typed languages suffice for soft typing.

We derive a soft type system like Λ_{SOFT} by starting with an appropriate static type system. In a static type system like Λ_{STATIC}, observe that adding a run-time check to a primitive operation with limited input domain yields an operation that accepts any input. For example, the primitive add1 has type $num \to num$, but the expression

```
(lambda (x)
  (if (number? x)
      (add1 x)
      (error "not a number"))),
```

which combines add1 with a run-time check, has polymorphic type $\forall\alpha.\,\alpha \to num$.[4] While the static type system we start with will not assign types to all programs, we choose a type system such that adding enough run-time checks to any Λ program will yield a typable program. Thus a soft type system consists of:

(i) a static type system that produces conclusions of the form $A \vdash e : \tau$, and

(ii) a transformation $e \Rightarrow e'$ that inserts run-time checks.

We combine these two systems as a deductive proof system that assigns both a transformed expression with run-time checks and a type to an expression. The proof system produces conclusions of the form $A \models\!\!\!\!> e \Rightarrow e' : \tau$, which means that e has run-time checks inserted to yield e' of type τ given type environment A (note the different turnstile $\models\!\!\!\!>$). The transformed expression e' is typable in the underlying static type system. Hence e' cannot reach a stuck program and Λ_{SOFT} is type safe.

Unfortunately, a soft type system as simple as Λ_{SOFT} assigns types that are too imprecise to provide much help to programmers. As we discuss in Chapter 6, such

[4]We subscribe to a view of types where quantification means intersection. In a statically typed language where types do not overlap, no procedure of type $\forall\alpha.\,\alpha \to num$ can use its argument because the intersection of all possible argument types is empty. In a type system where types overlap, the intersection can be populated, so there are procedures of type $\forall\alpha.\,\alpha \to num$ that make interesting use of their argument. This is the source of the expressive power of dynamically typed languages. Predicates like number?, which conventional statically typed languages do not provide, have type $\forall\alpha.\,\alpha \to bool$.

a soft type system all too frequently assigns to identifiers the uninformative type $\top$ or $\top \to \top$. The soft type system we develop in this thesis is a generalization of a static type system, as is Λ_{SOFT}. But rather than simply include a maximal type, our soft type system is based on an extension of the Hindley-Milner polymorphic type discipline with recursive types and restricted union types. Our type system assigns types that are precise enough to provide the benefits of type information, yet concise enough to be easy for programmers to interpret.

Chapter 3

A Soft Type System for Pure Scheme

In this chapter, we present a precise definition of a soft type system for a simple functional language called **Pure Scheme**. We begin by specifying the abstract syntax and operational semantics of **Pure Scheme**. We discuss the difficulty of assigning types to this language and develop a static type system that assigns types to a wide class of expressions. By generalizing this static type system to type all programs but insert some run-time checks, we obtain a soft type system for **Pure Scheme**.

3.1 Pure Scheme

As the first step in a formal description of our soft type system, we define an idealized, dynamically typed, call-by-value language embodying the essence of Scheme. Let Id be a denumerable set of *identifiers*, $Const$ be a set of *constants*, and $Prim \subseteq Const$ be a subset of constants called *primitives*. Let $x \in Id$, $c \in Const$, and $p \in Prim$.

> **Definition 3.1** *(Abstract Syntax of* **Pure Scheme***).* The expressions ($e \in Exp$) and *values* ($v \in Val$) of **Pure Scheme** are
>
> $$\begin{array}{rcll} e & ::= & v \;\big|\; (\textbf{ap}\ e_1\ e_2) \;\big|\; (\textbf{if}\ e_1\ e_2\ e_3) \;\big|\; (\textbf{let}\ ([x\ e_1])\ e_2) & (Exp) \\ & & \big|\; (\textbf{CHECK-ap}\ e_1\ e_2) & \\ v & ::= & c \;\big|\; x \;\big|\; (\textbf{lambda}\ (x)\ e). & (Val) \end{array}$$

Const includes *basic constants* (numbers, #t, #f, '()), *unchecked primitive operations* (add1, car, cons, *etc.*), and *checked primitive operations* (CHECK-add1, CHECK-car, *etc.*). The keywords **ap** and **CHECK-ap** introduce *unchecked* and *checked* applications, which we explain below. Procedures are constructed by **lambda** and take exactly one argument. The free identifiers, $FV(e)$, and bound identifiers of an expression are defined as usual, with **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 [7], 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; that is, programs have no free variables.

To incorporate run-time checks, **Pure Scheme** includes both *unchecked* and *checked* primitive operations and applications. Invalid applications of unchecked primitives, like (**ap** add1 #t) or (**ap** car '()), are meaningless. In an implementation, they can produce arbitrary results ranging from "core dump" to erroneous but apparently valid answers. Checked primitives are equivalent to their corresponding unchecked versions, except that invalid applications of checked primitives terminate execution with an error message. For example, (**ap** **CHECK-**add1 #t) yields an error message like "Error in add1: #t is not a number". Similarly, **ap** and **CHECK-ap** introduce unchecked and checked *applications* that are undefined (*resp.* yield an error message) when their first subexpression is not a procedure. For example, the expression (**ap** 1 2) is meaningless, while (**CHECK-ap** 1 2) produces an error message like "Error: 1 is not a procedure".

3.2 Operational Semantics

We use reduction semantics [18] to specify the operational behavior of **Pure Scheme** programs.

> **Definition 3.2** *(Semantics of* **Pure Scheme***).* The partial function $\longmapsto$: *Programs* $\rightharpoonup$ *Programs*, specified in Figure 3.1, defines a reduction semantics for **Pure Scheme** programs (neglecting pairs, which are easy to add).[1]

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,[2] 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 **lambda**-expressions by substitution. The notation $[x/v]e$ means the substitution of v for

[1] Recall that the notation $\rightharpoonup$ indicates a partial function.

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

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

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

Figure 3.1 Reduction Semantics for **Pure Scheme**

free occurrences of 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 $\#\mathsf{f}$. Rules 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. *ClosedVal* 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 CHECK-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, like (**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 (via **ap** or **CHECK-ap**) to a value for which it is not defined, *e.g.*, (**ap** add1 #t) or (**CHECK-ap** add1 #t), or when the first subexpression of an unchecked application is not a procedure, *e.g.*, (**ap** 1 2). We say that such a program is *stuck*.

Definition 3.3 *(Stuck Programs of* **Pure Scheme***).* The stuck programs of **Pure Scheme** are expressions of the form

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

Say that e diverges when there is an infinite reduction sequence $e \longmapsto e' \longmapsto e'' \longmapsto \ldots$ All closed expressions either (i) diverge, (ii) yield check, (iii) yield an answer that is a closed value, or (iv) become stuck.

> **Lemma 3.1** *(Uniform Evaluation).* For all closed expressions e, either e diverges, $e \longmapsto\!\!\!\!\rightarrow \mathsf{check}$, $e \longmapsto\!\!\!\!\rightarrow v$ where v is closed, or $e \longmapsto\!\!\!\!\rightarrow e'$ where $e' \in \mathit{Stuck}$.

Proof. The proof is a routine induction on the length of the reduction sequence, using case analysis on the structure of expression e. For a proof of a similar theorem, see Felleisen [17]. □

Type safe implementations of dynamically typed languages like **Pure 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 "machine crash" to erroneous but apparently valid answers.

3.3 Designing a Soft Type System

Designing a soft type system for **Pure 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 **Pure Scheme** should include union types and use the following rule to infer types for applications:

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

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 types except polymorphic types 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.* [4] have pursued this approach, but its computational complexity is significantly worse than Hindley-Milner typing. We elect instead to modify Hindley-Milner typing to accommodate union types and subtyping without compromising its practical efficiency.[3]

To combine union types and subtyping with Hindley-Milner polymorphism, we adapt an encoding Rémy developed for record subtyping [50, 52]. Our encoding permits many union types to be expressed as terms in a free algebra, as with conventional Hindley-Milner types. *Flag variables* enable polymorphism to encode subtyping as subset on union types. Types are inferred by a 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 our informal reasoning leads us to expect. Such a mismatch would be a serious liability for a static type system, as programs would be rejected by the type checker without a clear explanation. In a soft type system, this problem is not nearly as serious. Soft typed programs may contain apparently unmotivated run-time checks, but they can still be executed.

The next two sections (3.4 and 3.5) define a collection of static types and a static type inference system based on a variation of Rémy's encoding. Following that, Section 3.6 adapts this static type system to a soft type system for **Pure Scheme**. We discuss translating the inferred types into more readable *presentation types* in the next chapter.

3.4 Static Types

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

[3]Ordinary Hindley-Milner typing relies on simple unification of finite terms, which can be implemented in linear time [42, 47]. Our modification of Hindley-Milner typing requires unification of infinite terms, for which no linear algorithm is known. But practical implementations of simple unification do not use the linear time algorithms, and practical implementations of infinite unification are usually faster.

ing of the domain:

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

For **Pure Scheme**, all numbers inhabit the same partition because primitive operations like division and exponentiation can produce small integer, big integer, rational, real, or complex answers.[4] In a soft type system for an ML-like language where integers and reals are distinct kinds of numbers, 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 as the components of each ($\mathbb{D}_1$, $\mathbb{D}_2$) 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.)

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

$$\sigma, \tau \quad ::= \quad \kappa_1^{f_1} \vec{\sigma}_1 \cup \ldots \cup \kappa_n^{f_n} \vec{\sigma}_n \cup \left(\alpha \;\middle|\; \emptyset\right)$$

where

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

denotes a *flag*, and $\varphi \in \mathit{FlagVar}$ is a *flag variable*. 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

[4]It is feasible to further distinguish *exact* and *inexact* numbers as defined by the Scheme report [13].

set of possible types corresponding to the instances of the free flag variables. The following are some examples of types:

$num^{\boldsymbol{+}} \cup \not{\!\phi}$	means	"numbers;"
$num^{\boldsymbol{+}} \cup nil^{\boldsymbol{-}} \cup \not{\!\phi}$	means	"numbers;"
$num^{\boldsymbol{+}} \cup nil^{\boldsymbol{+}} \cup \not{\!\phi}$	means	"numbers or '();"
$num^{\boldsymbol{-}} \cup nil^{\boldsymbol{-}} \cup \not{\!\phi}$	means	"empty;"
$num^{\boldsymbol{+}} \cup nil^{\boldsymbol{-}} \cup \alpha$	means	"numbers or α but not '();"
$(\alpha \rightarrow^{\boldsymbol{+}} (true^{\boldsymbol{+}} \cup false^{\boldsymbol{+}} \cup \not{\!\phi})) \cup \not{\!\phi}$	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. Formally, we define the *tidy static types* of **Pure Scheme** as follows.

Definition 3.4 *(Static Types of* **Pure Scheme***).* The following class of grammars defines the static types (σ^X, τ^X) of **Pure Scheme**:

$$\begin{array}{rcl} \sigma^{\emptyset}, \tau^{\emptyset} & ::= & \alpha^{\emptyset} \;\big|\; \not{\!\phi}^{\emptyset} \;\big|\; (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^{\emptyset} \cup \tau^{\{\kappa\}} \;\big|\; \mu\alpha^{\emptyset}.\, \tau^{\emptyset} \\ \sigma^X, \tau^X & ::= & \alpha^X \;\big|\; \not{\!\phi}^X \;\big|\; (\kappa^f \sigma_1^{\emptyset} \ldots \sigma_n^{\emptyset})^X \cup \tau^{X \cup \{\kappa\}} \qquad\qquad (\kappa \notin X) \end{array}$$

where

$$f \;::=\; \boldsymbol{+} \;\big|\; \boldsymbol{-} \;\big|\; \varphi$$

is a *flag*; $\kappa \in Tag = \{num, true, false, nil, cons, \rightarrow\}$ is a *tag*; $\alpha^X, \beta^X \in TypeVar$ denote *type variables*; $\varphi \in FlagVar$ is a *flag variable*; $\nu \in (TypeVar \cup FlagVar)$ denotes a type or flag variable; and $X \in \mathbf{2}^{Tag}$ is a *label*.

Types represent regular trees with the tags κ constructing internal nodes. In a constructed type

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

the constructor κ has arity $n+2$: flag f, types $\sigma_1 \ldots \sigma_n$, and type τ are its arguments. The parentheses and union symbol $\cup$ are merely syntax to enhance readability. The *labels* X attached to types enforce tidiness by specifying sets of tags that are *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. The labels attached to types 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 [5]. The type $\mu\alpha.\,\tau$ binds α in τ; the usual renaming rules apply to the bound variable α, and we have $\mu\alpha.\,\tau = [\alpha/\mu\alpha.\,\tau]\tau$. Recursive types must be formally contractive, *i.e.*, phrases like $\mu\alpha.\,\alpha$ are not types. The type $\mu\alpha.\,nil^{+} \cup (cons^{+}\,\tau\,\alpha) \cup \phi$, which denotes proper lists[5] of τ, is a common recursive type.

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

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

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

It is easy to verify that this quotient preserves tidiness.

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

$$\Sigma \quad ::= \quad \forall \vec{\nu}.\, \tau. \qquad (\textit{Type Scheme})$$

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. $S\tau$ (resp. Sf) means the simultaneous replacement of every free variable in type τ (resp. 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

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

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

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

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

In our framework, we use polymorphism both to express conventional polymorphic types and to express subsets of tidy union types. The type scheme $\forall \varphi_1\varphi_2.\, num^{\varphi_1} \cup nil^{\varphi_2} \cup \emptyset$ may be instantiated to any type that denotes a subset of $num^{+} \cup nil^{+} \cup \emptyset$. There are four such types:

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

By using polymorphic flag variables for the inputs of primitive operations and procedures, we can simulate subtyping at applications while still using unification to equate the procedure's input type and the argument's type. A procedure with type scheme $\forall \varphi_1\varphi_2.\, (num^{\varphi_1} \cup nil^{\varphi_2} \cup \emptyset) \to^{+} \tau$ 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 $\mathbf{+}$ and $\mathbf{-}$. The application yields a result of type τ.

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 $\forall\alpha.\ num^{\mathbf{+}} \cup \alpha$. This may be instantiated to any type that is a superset of $num^{\mathbf{+}} \cup \emptyset$:

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

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

The function *TypeOf* maps the constants of **Pure 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 and $\mathbf{-}$ and $\emptyset$ for invalid inputs. Positive unions use $\mathbf{+}$ for "present" outputs and variables for "absent" fields. Figure 3.2 presents the types for some of the constants and unchecked primitive operations of Pure Scheme.

The type schemes of checked primitives are similar to those of unchecked primitives, except they never use $\mathbf{-}$ or $\emptyset$ since checked primitives accept all inputs. For example, primitives `CHECK-add1` and `CHECK-car` have the following type schemes:

$$\begin{array}{lcl} \textsf{CHECK-add1} & : & \forall\alpha_1\alpha_2\alpha_3\varphi.\,((num^{\varphi} \cup \alpha_3) \rightarrow^{\mathbf{+}} (num^{\mathbf{+}} \cup \alpha_1)) \cup \alpha_2 \\ \textsf{CHECK-car} & : & \forall\alpha_1\alpha_2\alpha_3\alpha_4\varphi.\,(((cons^{\varphi}\ \alpha_1\ \alpha_2) \cup \alpha_4) \rightarrow^{\mathbf{+}} \alpha_1) \cup \alpha_3. \end{array}$$

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 like `add1` that has type scheme

$$\forall\alpha_1\alpha_2\varphi.\,((num^{\varphi} \cup \emptyset) \rightarrow^{\mathbf{+}} (num^{\mathbf{+}} \cup \alpha_1)) \cup \alpha_2, \tag{3.1}$$

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

Figure 3.2 Static Types for Unchecked Constants

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

$$\forall\alpha_1\alpha_2.\, ((\mathit{num}^{+} \cup \emptyset) \to^{+} (\mathit{num}^{+} \cup \alpha_1)) \cup \alpha_2. \tag{3.2}$$

This type scheme does not have the trivial type $\emptyset$ as a subtype of its input type. Both (3.1) and (3.2) 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 8.1.2.

3.5 Static Type Checking

We use a deductive proof framework to define a *static type system* for **Pure Scheme**.

> **Definition 3.5** *(Static Type System for* **Pure Scheme***).* The axioms and inference rules in Figure 3.3 define a static type system for **Pure Scheme**. A judgment $A \vdash e : \tau$ constructed according to these rules is a *typing*. Program e has type τ if a deduction can be constructed according to these rules such that $\emptyset \vdash e : \tau$. Program e is untypable if there exists no τ' such that $\emptyset \vdash e : \tau'$.

In the figure, 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 scheme Σ. FV extends pointwise to type environments. When A is the empty map, we write simply $\vdash e : \tau$.

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

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

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

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

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

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

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

$$\mathit{Close}(\tau, A) \in \{\forall \vec{\nu}.\, \tau \mid \{\vec{\nu}\} \subseteq FV(\tau) - FV(A)\}$$

Figure 3.3 Static Typing for **Pure Scheme**

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 $\rightarrow$. As with the type schemes assigned to primitives (page 34), this typing avoids the problem of reverse flow (Section 8.1.2). Alternative typing rules for applications could use $\mathbf{+}$ rather than f.

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 a *typability* condition.

Definition 3.6 *(Typability Condition).* For every c, τ, τ', f, and v,

(i) 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 $\vdash v' : \tau$;

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

The first part of this condition 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. The second part of this condition ensures that only primitive operations, and not basic constants, may have functional type.

Given the typability condition, we can prove that our static type system is sound. Recall that Section 3.2 defines a reduction relation $\longmapsto\!\!\!\!\rightarrow$ for which every program either: (i) diverges, (ii) yields the error message check, (iii) yields an answer v, or (iv) gets *stuck* (Lemma 3.1). *Type soundness* ensures that typable programs yield answers of the expected type and do not get stuck.

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

Proof. The proof relies on two main lemmas. The first, *Subject Reduction*, states that evaluation preserves typing.

Lemma 3.2 *(Subject Reduction).* If $\vdash e_1 : \tau$ and $e_1 \longmapsto e_2$ then $\vdash e_2 : \tau$.

The proof of this lemma proceeds by induction over the structure of the derivation $\vdash e_1 : \tau$. For a full proof of *Subject Reduction*, see Appendix A.

The second main lemma required to prove *Type Soundness* states that stuck programs are not typable.

Lemma 3.3 *(Untypability of Stuck Programs).* If $e \in \mathit{Stuck}$ then there is no τ such that $\vdash e : \tau$.

Proof. We prove this lemma 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 $\mathbf{ap}_{\vdash}$, $A \vdash p : (\tau' \to^f \tau) \cup \emptyset$ and $A \vdash v : \tau'$ for some τ', f. But then by the typability condition, $\delta(p, v)$ is defined, which contradicts the premise of this case.

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

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

This completes the proof of Lemma 3.3. □

Now, from *Uniform Evaluation* (Lemma 3.1), either e diverges, $e \longmapsto\!\!\!\!\rightarrow \mathsf{check}$, $e \longmapsto\!\!\!\!\rightarrow v$ where v is closed, 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. □

3.6 Soft Type Checking

The preceding static type system can be used to statically type check **Pure 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.[6] A soft type checker automates this process.

As for the static type system, we use a deductive proof framework to define a *soft type system* for **Pure Scheme**.

> **Definition 3.7** *(Soft Type System for* **Pure Scheme***).* The axioms and inference rules in Figure 3.4 define a soft type system for **Pure Scheme** programs. A judgment $A \models e \Rightarrow e' : \tau$ constructed according to these rules is a *soft typing.* Program e receives run-time checks to yield program e' of type τ if a deduction can be constructed according to these rules such that $\emptyset \models e \Rightarrow e' : \tau$.

The rules in the figure compute a transformed expression in which some unchecked primitives and applications may be replaced by checked ones. The rules also determine a type for the transformed expression.

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* ($\tilde{\nu} \in \mathit{AbsVar} = (\mathit{AbsTypeVar} \cup \mathit{AbsFlagVar})$). 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 $\tilde{\varphi} \in \mathit{AbsFlagVar}$ or absent type variable $\tilde{\alpha} \in \mathit{AbsTypeVar}$. For example, *SoftTypeOf*(`add1`) is

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

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

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

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

[6]The same process cannot be used with statically typed languages like ML because the Hindley-Milner type discipline does not provide implicit union types. One must also introduce explicit definitions of tagged 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.

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

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

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

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

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

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

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

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

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

$$\mathit{SoftClose}(\tau, A) \ \in \ \left\{\forall\vec{\nu}.\,\tau \ \middle|\ \{\vec{\nu}\} \subseteq FV(\tau) - (FV(A) \cup \mathit{AbsVar})\right\}$$

Figure 3.4 Soft Typing for **Pure Scheme**

If an absent variable is instantiated to a non-empty 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) $true^{+} \cup \emptyset$. Since the type $true^{+} \cup \emptyset$ 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 **OKconst**$_{\models}$ and **OKap**$_{\models}$ in Figure 3.4 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, $Empty(s)$ is defined as

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

Rules **OKconst**$_{\models}$ and **OKap**$_{\models}$ require that *Empty* hold for that part of the unchecked primitive's input type corresponding to undefined inputs. Rules **const**$_{\models}$ and **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 **OKconst**$_{\models}$ and **OKap**$_{\models}$ over rules **const**$_{\models}$ and **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[7]

```
(let ([inc (lambda (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* naïvely

[7]As in the previous chapter, (**begin** e_1 e_2) abbreviates (**ap** (**lambda** (d) e_2) e_1) where $d \notin FV(e_2)$.

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 3.7 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 (*i*) all programs can be soft typed, and (*ii*) inserting run-time checks does not change the behavior of programs, other than to cause programs that would get stuck to yield check instead. To relate the behavior of source programs to the behavior of transformed programs, let $e \sqsubseteq e'$ mean that e' may have more checked operations that e, but e and e' are otherwise the same. Specifically, $\sqsubseteq$ is the reflexive, transitive, and compatible[8] closure of the following relation:

$$c \sqsubseteq_0 \mathsf{CHECK}\text{-}c$$

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

Then the following theorem establishes correctness for our soft type system.

Theorem 3.2 *(Correctness).*

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

(*ii*) For all e, e', τ such that $\models e \Rightarrow e' : \tau$:

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

Proof. The proof of the first part of this theorem proceeds by induction over the structure of typing derivations, using a strengthened induction hypothesis to accommodate terms with free identifiers. For a full proof, see Appendix A.

To prove the second part of *Correctness*, recall that evaluation has four possible outcomes. A program may (*i*) diverge, (*ii*) yield check, (*iii*) yield an answer v, or (*iv*)

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

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

Lemma 3.4 *(Simulation).* For $e_1 \sqsubseteq e_1'$:

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

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

For the forward direction of the second part of *Correctness*, from $\models e \Rightarrow e' : \tau$ we have $e \sqsubseteq e'$ by induction and case analysis of the soft typing rules in Figure 3.4. By induction with the first part of Simulation:

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

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

Let $\tilde{S}$ be the substitution with domain $AbsVar$ that takes all absent type variables to $\emptyset$ and all absent flag variables to $-$. We can show that e' is typable in the static type system of Section 3.4.

Lemma 3.5 *(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 height of the deduction $A \models e \Rightarrow e' : \tau$. A proof of this lemma may be found in Appendix A. Given this lemma, *Type Soundness* ensure that e' cannot get stuck.

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

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

This completes the proof of *Correctness*. $\square$

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

Corollary 3.1 *(Safety).* For all e, e', τ such that $\models e \Rightarrow e' : \tau$, program e' does not get stuck.

3.7 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.

3.7.1 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 following expression:

```
(let ([inc (lambda (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 non-empty 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 3.5. 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:

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

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

Figure 3.5 Program Generating Spurious Run-time Checks

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 $true^{+} \cup \tilde{\alpha}_4$. Procedure inc now has type scheme:

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

hence the application (**ap** inc x) forces x to have type $num^{+} \cup true^{+} \cup \tilde{\alpha}_4$. Thus 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 non-empty. 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.[9] To permit generalization of absent variables, we replace rules $\mathbf{let}_{\models\!\!\Rightarrow}$ and $\mathbf{id}_{\models\!\!\Rightarrow}$

[9] A precise formulation requires assigning unique names to each binding occurrence of a variable within a deduction tree.

with the following rules:

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

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

The new $\textbf{let}_2$ rule uses *Close* rather than *SoftClose* to generalize variables. The new $\textbf{var}_2$ rule records the instances of absent variables in Ψ. $S|_{\mathit{AbsVar}}$ means S restricted to absent variables. Finally, we redefine *Empty* as follows:

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

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 3.5 again *with* the application (**ap** `inc` #t). Procedure `inc` now has type scheme:

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

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

$$\Psi = \{\{\tilde{\alpha}_3/\tilde{\alpha}_5\}, \{\tilde{\alpha}_3/\mathit{true}^{\mathbf{+}} \,\cup\, \tilde{\alpha}_4\}\}.$$

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

3.7.2 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 $+$ 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:[10]

(**letrec** ([f (**lambda** (x) (**ap** f (**ap** sub1 x)))])
  f)

procedure f has soft type scheme:

$$\forall \alpha_2. ((num^{+} \cup \tilde{\alpha}_1) \rightarrow^{+} \alpha_2) \cup \tilde{\alpha}_3. \tag{3.3}$$

(The absent variables $\tilde{\alpha}_1$ and $\tilde{\alpha}_3$ may be generalized if the extension from Section 3.7.1 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 $+$-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 non-procedural value:

(**letrec** ([f (**lambda** (x) (**ap** f (**ap** sub1 x)))])
  (**if** *e* f #t)) (3.4)

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

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

now includes $true^{+}$.

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

$$\forall \varphi \alpha_2 \alpha_3. ((num^{\varphi} \cup \tilde{\alpha}_1) \rightarrow^{+} \alpha_2) \cup \alpha_3. \tag{3.5}$$

denotes the same set of values as type 3.3. To see this, convert both to their equivalent static types:

$$\forall \alpha_2. ((num^{+} \cup \emptyset) \rightarrow^{+} \alpha_2) \cup \emptyset \qquad \text{(static equivalent of 3.3)}$$
$$\forall \varphi \alpha_2 \alpha_3. ((num^{\varphi} \cup \emptyset) \rightarrow^{+} \alpha_2) \cup \alpha_3 \qquad \text{(static equivalent of 3.5)}$$

[10]We use the standard method of typing **letrec**-expressions. See Chapter 5 for more details.

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 3.3 with type 3.5 in assigning a type to f. Unlike type 3.3, this new type has no $+$-flags in negative positions, nor absent type variables in positive positions. By replacing the type for f in example 3.4 with type 3.5 when generalizing f's type, example 3.4 will no longer require an unnecessary run-time check.

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

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

The function $\mathcal{T}[\![\cdot]\!]$, defined in Appendix B, yields the set of values that its argument type denotes. The second antecedent of this rule 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.

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

(ii) 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 7.2 presents an example that illustrates the effect of this extension for a real program.

3.8 Inserting Errors

The soft type systems described above insert run-time checks at primitive operations that *might* lead to errors. The systems insert a run-time check whenever type inference indicates that an invalid argument may arise at the input of a primitive. We can also use type inference to determine whether a primitive operation is applied to a *valid* argument. If type inference indicates that a primitive operation 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*, we specify that every unchecked primitive operation c has both an ordinary checked version CHECK-c and an error version ERROR-c. The interpretation function δ is defined and yields check for any argument of an ERROR-c primitive:

$$\delta(\mathsf{ERROR}\text{-}c, v) = \mathsf{check} \quad \text{for all } v.$$

We also add a new form of expression for procedure applications that are errors:

$$e \ ::= \ \ldots \ \big| \ (\textbf{ERROR-ap}\ e_1\ e_2)$$

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 further extend the soft type system of the previous subsection. We track instantiations of all variables, rather than just absent variables, by including all instantiating substitutions in Ψ:

$$\frac{\tau \prec_S A(x) \qquad S \in \Psi}{A \mathrel{\triangleright\!\!\!\!\models} x \Rightarrow x : \tau} \qquad (\mathbf{var}_3)$$

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 like add1 that has type scheme:

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

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 following program:

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

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

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

φ is empty and add1 receives an *error*:

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

Inserting *errors* for other primitives and for applications of non-procedures proceeds in a similar manner.

3.9 Implementing Type Inference

As our type inference method extends the Hindley-Milner system with union types and recursive types, we simply adapt conventional algorithms for Hindley-Milner type inference to our system. Rémy [53] describes the basic algorithms for unification, generalization, and instantiation of types in detail.

Chapter 4

Types for Programmers

The soft type system developed in the previous chapter infers relatively precise types for Pure Scheme programs. The inferred types enable effective elimination of run-time checks and optimization of data representations. But these types are difficult for programmers to interpret due to the notational complexity introduced by (i) flags and (ii) type variables used to encode subtyping. In this chapter, we define simpler *presentation* types that are easier for programmers to interpret. We present a translation from the *internal* types inferred by our soft type system to presentation types.

4.1 Presentation Types

Our presentation types include recursive types and tidy union types, just as internal types do.

Definition 4.1 *(Presentation Types for* **Pure Scheme***).* The *presentation types* (T) for **Pure Scheme** are

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

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

$$\begin{array}{rcl} U & ::= & B \ \big| \ X \ \big| \ (+\ B_1 \ldots B_n \left[N_1 \ldots N_m\ X \right]) \\ B & ::= & \mathit{num} \ \big| \ \mathit{true} \ \big| \ \mathit{false} \ \big| \ \mathit{nil} \ \big| \ (\mathit{cons}\ U_1\ U_2) \ \big| \ (U_1 \rightarrow U_0) \\ N & ::= & (\mathit{not\ num}) \ \big| \ (\mathit{not\ true}) \ \big| \ (\mathit{not\ false}) \ \big| \ (\mathit{not\ nil}) \\ & & \big| \ (\mathit{not\ cons}) \ \big| \ (\mathit{not} \rightarrow). \end{array}$$

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

$X_1 \dots X_n$ in $U_1 \dots U_n$ 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)* (that is, *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 3.4). 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 *(+* $B_1 \dots B_n$*)*. When a union type includes a type variable, the type variable's universe implicitly excludes any types constructed from the tags of $B_1 \dots B_n$. Place holders serve to further constrain the universe of a type variable, without increasing the type. If a union type includes place holders preceding a type variable, as in the type

(+ $B_1 \dots B_n$ $N_1 \dots N_m$ X*)*,

then the universe for the type variable X also excludes any types constructed from the tags of $N_1 \dots N_m$. 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* matches 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. Procedure f:

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

(define g2
  (lambda (x)
    (if (null? x)
        1
        (if (number? x)
            '()
            x))))
```

both have type:

$$\textit{((+ num nil X1) -> (+ num nil X1))}. \tag{4.1}$$

Such shared type variables indicate a modicum of dependence of the result on the input. The interpretation of this type is:

(i) Given a number, a procedure of type 4.1 may return a number or the empty list.

(ii) Given an empty list, a procedure of type 4.1 may return a number or the empty list.

(iii) Given an input of type *X1* (which excludes *num* and *nil*), a procedure of type 4.1 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 illustrates a presentation type that involves a union type, a recursive type, place holders, and a shared type variable:

```
(define flatten
  (lambda (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 non-empty leaves of the tree. The leaves are any values in the input other than pairs or the empty list. We explain this example further in Chapter 7.

Following are the types for a few well-known Scheme functions. Several of these types use the next chapter's extended procedure types *($U_1 \ldots U_n$ -> U_0)* of arity n.

map : *((X1 –> X2) (list X1) –> (list X2))*

member : *(X1 (list X1) –> (+ false (cons X1 (list X1))))*

lastpair : *(***rec** *([Y1 (+ (cons X1 Y1) X2)])*
((cons X1 Y1) –> (cons X1 (+ (not cons) X2))))

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

Chapter 7 contains additional examples of presentation types.

4.2 Displaying Presentation Types

To translate inferred internal types into presentation types, we must eliminate (i) flags and (ii) 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. The definition assumes that all bound variables in the typing differ and are distinct from free variables.

Definition 4.2 *(Useless Variables).* With respect to soft typing $\models e \Rightarrow e' : \tau$, type or flag variable ν is *useless* if:

(i) ν is never generalized; or

(ii) ν is an absent variable; or

(iii) 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 ordinary prime types. Components with flag $-$ translate to place holders (*not* κ), or are dropped entirely if the union ends in $\emptyset$.

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

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

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, and φ_3 and φ_4 because they do not occur negatively in the above type. Replacing the useless variables with $\emptyset$ and $-$ as appropriate, and replacing the remaining flag variables φ_1, φ_2 with $+$ yields:

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

Changing syntax, we have:

(**rec** *([Y1 (+ nil (cons Y1 Y1) X1)]*
[Y2 (+ nil (cons (+ (not nil) (not cons) X1) Y2))])
(Y1 –> Y2))

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:

$$\forall\varphi_1\varphi_2.\,(true^{\varphi_1}\ \cup\ false^{\varphi_2}\ \cup\ \emptyset) \rightarrow^{+} (true^{\varphi_1}\ \cup\ false^{\varphi_2}\ \cup\ \emptyset)\ \cup\ \emptyset \tag{4.2}$$

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 dependences 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 prime types. For example, the above internal type 4.2 denotes the conjunction

((+ true false) –> (+ true false))
and *(true –> true)*
and *(false –> false)*
and *((+) –> (+))*.

(Note that *(+)* is the empty presentation type corresponding to $\emptyset$.) Cartwright and Fagan [10] 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 4.2 would be printed as

(true –> true) **and** *(false –> 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 4.2. 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 4.2 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 3.3). Fortunately, this imperfection does not seem to matter for practical programming.

Chapter 5

Beyond Pure Scheme

Practical programming languages extend our idealized language **Pure Scheme** with facilities like assignment, non-local control operators, and data definition. These facilities increase the expressiveness of a programming language [17]. Other facilities like pattern matching and records permit more precise type assignment. And other features like macro definition provide important syntactic convenience. In this chapter, we extend our soft type system for **Pure Scheme** to encompass many features of realistic programming languages. We pay particular attention to the features of R4RS Scheme and discuss their incorporation in our soft type system for Scheme.

5.1 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!. In this section, we indicate how to extend **Pure Scheme** to handle Scheme's various forms of assignment. To simplify a formal treatment of assignment, in the first subsection we consider only a simple assignable object called a *box*. We define the semantics and typing rules of **Box Scheme**, an extension of **Pure Scheme** that includes boxes. The second subsection discusses Scheme's various assignment forms.

5.1.1 Box Scheme

Box Scheme extends the syntax of **Pure Scheme** with three unchecked operations on boxes and two checked operations.

Definition 5.1 *(Abstract Syntax of* **Box Scheme***).* The expressions and values of **Box Scheme** extend those of **Pure Scheme** (Definition 3.1) to

$$\begin{aligned} e \ ::= \ & \ldots \mid (\textbf{box}\ e) \mid (\textbf{unbox}\ e) \mid (\textbf{set-box!}\ e_1\ e_2) \\ & \mid (\textbf{CHECK-unbox}\ e) \mid (\textbf{CHECK-set-box!}\ e_1\ e_2). \end{aligned}$$

The expression (**box** e) allocates a new box containing the value of e and returns the box. The expression (**unbox** e) requires e to evaluate to a box and extracts the contents of this box. The expression (**set-box!** e_1 e_2) requires e_1 to evaluate to a box and replaces the contents of this box with the value of e_2. Since we are not interested in the result returned by **set-box!**, all uses of **set-box!** will return the special value #void. (**CHECK-unbox** e) is similar to (**unbox** e) but performs a run-time check to ensure that its argument evaluates to a box. Similarly, (**CHECK-set-box!** e_1 e_2) performs a run-time check to ensure that e_1 evaluates to a box. There is no checked form of **box**-expression because there is nothing to check—**box** works with any value.

To extend the reduction semantics for **Pure Scheme** (Figure 3.1) to boxes, we must include the current contents of a program's allocated boxes in the representation of intermediate states of evaluation.

> **Definition 5.2** *(Evaluation States of* **Box Scheme***).* A **Box Scheme** evaluation state has the syntactic representation
>
> $$\theta \,!\, e$$
>
> where
>
> $$\theta \ ::= \ \langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.$$

The expression e represents the control state of the computation. The sequence of pairs θ represents the contents of the store. The first component of each pair is a box name and the second component is the value stored in that box. The phrase $\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle \,!\, e$ binds $x_1, \ldots, x_n$ in $v_1, \ldots, v_n$ and e, permitting recursive bindings like **letrec**. While θ is syntactically a sequence of pairs, we treat θ as a finite function with domain $\{x_1, \ldots, x_n\}$. That is, we disregard the order of pairs and require that the box names be unique. As with expressions, we identify states that differ by a consistent renaming of bound identifiers. Programs are closed expressions with an empty initial store ($\emptyset \,!\, e$).

Extending the existing reductions of **Pure Scheme** (Figure 3.1) to apply to **Box Scheme** requires two changes. First, we must extend the definition of evaluation contexts to include the new expression forms:

$$\begin{aligned} E \ ::= \ \ldots \ &\big|\ (\textbf{box}\ E) \ \big|\ (\textbf{unbox}\ E) \ \big|\ (\textbf{set-box!}\ E\ e) \ \big|\ (\textbf{set-box!}\ v\ E) \\ &\big|\ (\textbf{CHECK-unbox}\ E) \\ &\big|\ (\textbf{CHECK-set-box!}\ E\ e) \ \big|\ (\textbf{CHECK-set-box!}\ v\ E) \end{aligned}$$

Second, we adapt each reduction rule from Figure 3.1 by including a store:

$$\theta \,!\, E[(\textbf{ap}\ (\textbf{lambda}\ (x)\ e)\ v)] \ \longmapsto\ \theta \,!\, E[e[x/v]] \qquad (\beta_{\mathrm{v}})$$

$$\theta \,!\, E[(\textbf{CHECK-ap}\ (\textbf{lambda}\ (x)\ e)\ v)] \ \longmapsto\ \theta \,!\, E[e[x/v]] \qquad (\textit{check-}\beta_{\mathrm{v}})$$

$$\textit{etc.}$$

The extended reductions leave the store unchanged.

Figure 5.1 shows the additional reductions for **Box Scheme**. The reduction *box* expands the domain of the current store θ by picking a new box name not in the domain of θ. The *un* and *check-un*$_1$ reductions extract the contents of a box by looking up its name in the store to find the corresponding value. Reduction *check-un*$_2$ aborts program execution if v is not an identifier. Note that evaluation contexts E do not bind identifiers, so if the subexpression of an **unbox**- or **CHECK-unbox**-expression is an identifier, it must be bound in the store θ. The *set* and *check-set*$_1$ reductions change the value recorded in the store for the named box. Reduction *check-set*$_2$ aborts program execution if the first subexpression of a **CHECK-set-box!**-expression is not an identifier.[1]

As with **Pure Scheme**, programs can become stuck when an unchecked operation is applied to an invalid argument. With **Box Scheme**, there are three new ways programs can become stuck:

$$\textit{Stuck} \ = \ \left\{ \begin{array}{ll} \theta \,!\, E[(\textbf{ap}\ p\ v)] & \text{where } \delta(p,v) \text{ is undefined} \\ \theta \,!\, E[(\textbf{CHECK-ap}\ p\ v)] & \text{where } \delta(p,v) \text{ is undefined} \\ \theta \,!\, E[(\textbf{ap}\ c\ v)] & \text{where } c \notin \textit{Prim} \\ \hline \theta \,!\, E[(\textbf{ap}\ x\ v)] & \\ \theta \,!\, E[(\textbf{unbox}\ v)] & \text{where } v \notin \textit{Id} \\ \theta \,!\, E[(\textbf{set-box!}\ v_1\ v_2)] & \text{where } v_1 \notin \textit{Id} \end{array} \right\} \begin{array}{l} \text{(old)} \\ \\ \text{(new)} \end{array}.$$

As before, a program that applies (via **ap** or **CHECK-ap**) an unchecked primitive to invalid arguments is stuck, as is a program that applies a basic constant via **ap**. In

[1] This semantics for boxes accumulates all boxes ever allocated during a program's execution in the store. If desired, we can add a reduction that can be applied at any time to garbage-collect unreferenced boxes:

$$\theta_1\theta_2 \,!\, E[e] \ \longmapsto\ \theta_1 \,!\, E[e] \qquad \text{if } \mathrm{Dom}(\theta_1) \not\pitchfork \mathrm{Dom}(\theta_2) \text{ and } \theta_1 \,!\, E[e] \text{ is closed.}$$

$$\begin{array}{rcllr}
\theta\,!\,E[(\mathbf{box}\ v)] & \longmapsto & \theta\langle x,v\rangle\,!\,E[x] & x\notin \mathrm{Dom}(\theta) & (\mathit{box}) \\
\theta\langle x,v\rangle\,!\,E[(\mathbf{unbox}\ x)] & \longmapsto & \theta\langle x,v\rangle\,!\,E[v] & & (\mathit{un}) \\
\theta\langle x,v_1\rangle\,!\,E[(\mathbf{set\text{-}box!}\ x\ v_2)] & \longmapsto & \theta\langle x,v_2\rangle\,!\,E[\#\mathsf{void}] & & (\mathit{set}) \\
\theta\langle x,v\rangle\,!\,E[(\mathbf{CHECK\text{-}unbox}\ x)] & \longmapsto & \theta\langle x,v\rangle\,!\,E[v] & & (\mathit{check\text{-}un}_1) \\
\theta\,!\,E[(\mathbf{CHECK\text{-}unbox}\ v)] & \longmapsto & \mathsf{check} & \text{if } v\notin \mathit{Id} & (\mathit{check\text{-}un}_2) \\
\theta\langle x,v_1\rangle\,!\,E[(\mathbf{CHECK\text{-}set\text{-}box!}\ x\ v_2)] & \longmapsto & \theta\langle x,v_2\rangle\,!\,E[\#\mathsf{void}] & & (\mathit{check\text{-}set}_1) \\
\theta\,!\,E[(\mathbf{CHECK\text{-}set\text{-}box!}\ v_1\ v_2)] & \longmapsto & \mathsf{check} & \text{if } v_1\notin \mathit{Id} & (\mathit{check\text{-}set}_2)
\end{array}$$

Figure 5.1 Additional Reductions for **Box Scheme**

addition, any program that uses **ap** to apply a box, uses **unbox** on a non-box value, or uses **set-box!** with a non-box value, is stuck. It is straightforward to extend the *Uniform Evaluation* lemma to show that all closed **Box Scheme** expressions either (i) diverge, (ii) yield check, (iii) yield an answer that is a closed value, or (iv) become stuck.

Extending our type system for **Pure Scheme** to boxes requires introducing type constructors for boxes and the special value #void. The type constructor for a box, $(box^f\,\sigma)$, includes the type of the elements of the box, σ. The type constructor for #void is simply $void^f$. Figure 5.2 presents the additional static type inference rules for box operations.

If our type system did not include polymorphic **let**-expressions, the preceding changes would suffice to type **Box Scheme**. But as many authors have noted, naïvely combining Hindley-Milner polymorphism and assignment leads to an unsound type system [62]. Many solutions to this problem have been proposed [6, 14, 22, 29, 35, 38, 57, 60, 61, 62]. We adapt our own solution [62], which is the simplest of all. This solution restricts polymorphism to syntactic values. That is, we replace **Pure Scheme**'s static type inference rule for **let**-expressions (Figure 3.3) with the following two inference rules.

$$\frac{A \vdash e : \tau_1}{A \vdash (\textbf{box}\ e) : (box^{\boldsymbol{+}}\tau_1) \cup \tau_2} \qquad (\textbf{box}_{\vdash})$$

$$\frac{A \vdash e : (box^{f}\tau_1) \cup \phi}{A \vdash (\textbf{unbox}\ e) : \tau_1} \qquad (\textbf{un}_{\vdash})$$

$$\frac{A \vdash e : (box^{f}\tau_1) \cup \tau_2}{A \vdash (\textbf{CHECK-unbox}\ e) : \tau_1} \qquad (\textbf{CHECK-un}_{\vdash})$$

$$\frac{A \vdash e_1 : (box^{f}\tau_1) \cup \phi \qquad A \vdash e_2 : \tau_1}{A \vdash (\textbf{set-box!}\ e_1\ e_2) : void^{\boldsymbol{+}} \cup \tau_2} \qquad (\textbf{set}_{\vdash})$$

$$\frac{A \vdash e_1 : (box^{f}\tau_1) \cup \tau_3 \qquad A \vdash e_2 : \tau_1}{A \vdash (\textbf{CHECK-set-box!}\ e_1\ e_2) : void^{\boldsymbol{+}} \cup \tau_2} \qquad (\textbf{CHECK-set}_{\vdash})$$

Figure 5.2 Static Typing for **Box Scheme**

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

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

The first rule assigns a polymorphic type 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.

We can prove *Type Soundness* for **Box Scheme** by extending our proof of *Type Soundness* for **Pure Scheme**. Extending the proof requires introducing a type inference rule for programs since programs now include stores:

$$\frac{\begin{array}{l}[x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_n] \vdash v_1 : \tau_1 \\ \qquad\qquad \vdots \\ [x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_n] \vdash v_n : \tau_n \\ [x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_n] \vdash e : \tau\end{array}}{\vdash \langle x_1, v_1 \rangle \ldots \langle x_n, v_n \rangle\,!\,e : \tau}$$

The details of this proof are a straightforward adaptation and simplification of our type soundness proof [64] for Tofte's method of typing boxes.

The soft type inference rules for **Box Scheme** can be derived from the static type inference rules in the same manner as the soft typing rules for **Pure Scheme** are derived from its static typing rules. Figure 5.3 presents the additional soft typing rules for boxes and the modified rule for **let**-expressions. Given *Type Soundness*, the proofs for an extended *Correctness* theorem are simple extensions of those for **Pure Scheme**.

We discuss the general advantages of our method of integrating polymorphism and assignment elsewhere [62]. 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, we strongly suspect that the additional notations introduced by these systems would preclude decoding types into a simple presentation type language. For example, Tofte's system [60] 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.

$$\frac{A \models e \Rightarrow e' : \tau_1}{A \models (\mathbf{box}\ e) \Rightarrow (\mathbf{box}\ e') : (box^{+}\tau_1) \cup \tau_2} \qquad (\mathbf{box}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (box^{f}\tau_1) \cup \tilde{\tau}_2 \qquad Empty(\tilde{\tau}_2)}{A \models (\mathbf{unbox}\ e) \Rightarrow (\mathbf{unbox}\ e') : \tau_1} \qquad (\mathbf{OKun}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (box^{f}\tau_1) \cup \tilde{\tau}_2}{A \models (\mathbf{unbox}\ e) \Rightarrow (\mathbf{CHECK\text{-}unbox}\ e') : \tau_1} \qquad (\mathbf{un}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (box^{f}\tau_1) \cup \tau_2}{A \models (\mathbf{CHECK\text{-}unbox}\ e) \Rightarrow (\mathbf{CHECK\text{-}unbox}\ e') : \tau_1} \qquad (\mathbf{CHECK\text{-}un}_{\models})$$

$$\frac{A \models e_1 \Rightarrow e_1' : (box^{f}\tau_1) \cup \tilde{\tau}_2 \qquad A \models e_2 \Rightarrow e_2' : \tau_1 \qquad Empty(\tilde{\tau}_2)}{A \models (\mathbf{set\text{-}box!}\ e_1\ e_2) \Rightarrow (\mathbf{set\text{-}box!}\ e_1'\ e_2') : void^{+} \cup \tau_3} \qquad (\mathbf{OKset}_{\models})$$

$$\frac{A \models e_1 \Rightarrow e_1' : (box^{f}\tau_1) \cup \tilde{\tau}_2 \qquad A \models e_2 \Rightarrow e_2' : \tau_1}{A \models (\mathbf{set\text{-}box!}\ e_1\ e_2) \Rightarrow (\mathbf{CHECK\text{-}set\text{-}box!}\ e_1'\ e_2') : void^{+} \cup \tau_3} \qquad (\mathbf{set}_{\models})$$

$$\frac{A \models e_1 \Rightarrow e_1' : (box^{f}\tau_1) \cup \tau_2 \qquad A \models e_2 \Rightarrow e_2' : \tau_1}{A \models (\mathbf{CHECK\text{-}set\text{-}box!}\ e_1\ e_2) \Rightarrow (\mathbf{CHECK\text{-}set\text{-}box!}\ e_1'\ e_2') : void^{+} \cup \tau_3} \qquad (\mathbf{CHECK\text{-}set}_{\models})$$

$$\frac{A \models v_1 \Rightarrow v_1' : \tau_1 \qquad A[x \mapsto SoftClose(\tau_1, A)] \models e_2 \Rightarrow e_2' : \tau_2}{A \models (\mathbf{let}\ ([x\ v_1])\ e_2) \Rightarrow (\mathbf{let}\ ([x\ v_1'])\ e_2') : \tau_2} \qquad (\mathbf{letval}_{\models})$$

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

$+$ {**Pure Scheme** soft typing rules (Figure 3.4) – $\mathbf{let}_{\models}$}

Figure 5.3 Soft Typing for **Box Scheme**

5.1.2 Assignment in Scheme

As mentioned earlier, Scheme provides assignment to pairs, vectors, and identifiers.

Adapting our formal treatment of boxes to pairs and vectors is straightforward. The primitive operations cons and vector construct pairs and vectors. Applications of cons and vector are not values and hence inhibit polymorphism. But this solution treats all pairs as mutable, and many pairs used in Scheme programs are immutable. Inhibiting polymorphism for immutable pairs also inhibits subtyping, which can result in less precise type assignment. Consider the following program:

```
(let ([x (cons 1 '())])
  (length x)
  (car x))
```

Since (cons 1 '()) is not a value, x is not assigned a polymorphic type. The application of length causes x to have type *(list num)*, *i.e.*

$$\mu\alpha_1.\, nil^{+} \cup (cons^{+}(num^{+} \cup \alpha_2)\,\alpha_1) \cup \tilde{\alpha}_3.$$

Hence car requires a run-time check. As the pairs constructed by the two applications of cons are never modified in this program, it would be safe to assign to x the type scheme *(cons num nil)*, *i.e.*

$$\forall\alpha_1\alpha_2\alpha_3.\,(cons^{+}(num^{+} \cup \alpha_1)\,(nil^{+} \cup \alpha_2)) \cup \alpha_3.$$

Since this type scheme is polymorphic in α_3 and would be instantiated independently for each use of x, car would require no run-time check. (Since α_3 is polymorphic, the application of length "rolls up" only the first instance of x's type.)

If we were able to determine which pairs are never modified in a program, we could treat as values applications of cons that construct immutable cells. For our prototype, we use the following expedient solution. If the program contains no occurrences of set-car! or set-cdr!, all pairs are assumed to be immutable. Applications of cons are treated as syntactic values. Otherwise, all pairs are considered mutable, and applications of cons are not values.

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 non-assignable according to whether they appear in the first position of a **set!**-expression. Assignable identifiers are not considered values for typing **let**-expressions. That is, in the following program:

(**set!** y 1)
$\vdots$
(**let** ([x y])
...)

the type of y must not be generalized in assigning a type scheme to x as y is an assignable identifier.

Scheme also allows assignment via **set!** to primitive operations like car. To permit assignment to primitives, we treat programs as being preceded by definitions of the primitive operators:

(**define** car *primitive-car*)
(**define** cdr *primitive-cdr*)
$\vdots$
user's program

With the interpretation of Scheme programs as **letrec**-expressions in Section 5.5, our treatment of **set!** handles assignment to primitives.

5.2 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 [13]. This operator provides a powerful form of non-local control that can be used to define exceptions, build back-tracking algorithms, schedule multiple threads of control, etc. In this section, we extend **Box Scheme** to **Imperative Scheme**, which includes both boxes and first-class continuations.

We extend **Box Scheme** to provide access to the current continuation by adding one unchecked expression form and one checked expression form.

Definition 5.3 *(Abstract Syntax of* **Imperative Scheme***).* The expressions and values of **Imperative Scheme** extend those of **Box Scheme**, given in Definition 5.1, to

$$e ::= \ldots \mid (\textbf{call/cc}\ e) \mid (\textbf{CHECK-call/cc}\ e).$$

The expression (**call/cc** e) requires e to evaluate to a procedure and applies this procedure to an abstraction of the continuation of the **call/cc**-expression. This *continuation* is packaged as a procedure. When applied to an argument v, this continuation procedure will transfer control back to **call/cc**-expression, which will return v. (**CHECK-call/cc** e) is similar to (**call/cc** e), but performs a run-time check to ensure that e evaluates to a procedure. Scheme's call-with-current-continuation procedure is defined as:

```
(define call-with-current-continuation
  (lambda (x)
    (call/cc x)))
```

To specify the semantics of **call/cc**-expressions, we add an additional expression form that is not in the surface language available to programmers.

$$e \ ::= \ \ldots \ \big| \ (\mathcal{A}\ e)$$

An $\mathcal{A}$-expression ("abort-expression") merely discards the surrounding program context and evaluates e as the final answer of the program.

The existing reductions of **Box Scheme** require no modifications other than the extension of evaluation contexts:

$$E \ ::= \ \ldots \ \big| \ (\textbf{call/cc}\ E)$$

Two additional reductions for **call/cc**- and $\mathcal{A}$-expressions are necessary:

$$\theta\,!\,E[(\textbf{call/cc}\ v)] \ \longmapsto\ \theta\,!\,E[v\ (\textbf{lambda}\ (x)\ (\mathcal{A}\ E[x]))] \qquad (\textit{call/cc})$$

$$\theta\,!\,E[(\mathcal{A}\ e)] \ \longmapsto\ \theta\,!\,e \qquad (\mathcal{A})$$

A *call/cc* reduction copies the program's continuation E and constructs a procedure (**lambda** (x) $(\mathcal{A}\ E[x])$) that represents this continuation. The constructed procedure uses an $\mathcal{A}$-expression to terminate execution when the copied continuation finishes. When an $\mathcal{A}$-expression becomes active, the $\mathcal{A}$-reduction simply discards its evaluation context E.

The static type inference rules for **Imperative Scheme** are those of **Box Scheme** with three additional rules for the new expression forms.

$$\frac{A \vdash e : (((\tau_1 \rightarrow^{+} \tau_2) \ \cup \ \tau_3) \rightarrow^{f} \tau_1) \ \cup \ \phi}{A \vdash (\textbf{call/cc}\ e) : \tau_1} \qquad (\textbf{call/cc}_{\vdash})$$

$$\frac{A \vdash e : (((\tau_1 \rightarrow^{+} \tau_2) \ \cup \ \tau_3) \rightarrow^{f} \tau_1) \ \cup \ \tau_4}{A \vdash (\textbf{CHECK-call/cc}\ e) : \tau_1} \qquad (\textbf{CHECK-call/cc}_{\vdash})$$

$$\frac{A \vdash e : \tau_1}{A \vdash (\mathcal{A}\ e) : \tau_2} \qquad (\boldsymbol{\mathcal{A}}_{\vdash})$$

The expression (**call/cc** e) requires that e be a procedure, hence the antecedent for rule $\textbf{call/cc}_{\vdash}$ requires e have a type of the shape $(\ldots \rightarrow^{f} \tau_1) \ \cup \ \phi$. The procedure to which e evaluates must accept a continuation, which is represented as a procedure, and may accept anything else. Hence the input type for e is $(\tau_1 \rightarrow^{+} \tau_2) \ \cup \ \tau_3$. The continuation's input type τ_1 matches the type of the expression (**call/cc** e) since a value passed to the continuation becomes the result of the **call/cc**-expression. The continuation's result type τ_2 is arbitrary since the continuation never returns. The **CHECK-call/cc**-expression has a more permissive typing rule than the **call/cc**-expression because it includes a run-time check to ensure that e evaluates to a procedure. An $\mathcal{A}$-expression may have any type, regardless of the type of its subexpression, because it never returns a value to its surrounding context.

Harper and Lillibridge [23] discovered that naïvely combining first-class continuations with polymorphism leads to an unsound type system, just as naïvely combining assignment and polymorphism does. Fortunately, the same solution of restricting polymorphism to values works for both assignment and first-class continuations [64]. By building **Imperative Scheme** on top of **Box Scheme**, we have already incorporated this restriction in our type system for **Imperative Scheme**.

Because $\mathcal{A}$-expressions can return a value of any type, proving type soundness for **Imperative Scheme** is not as simple as for **Pure Scheme** or **Box Scheme**. The problem is that subject reduction does not hold for the $\mathcal{A}$ reduction. For example, the expression

(**if** e 1 ($\mathcal{A}$ #t))

has type *num* according to the above static typing rules, but returns #t if e evaluates to #f. To prove that the static type system for **Imperative Scheme** is sound, we exploit the fact that $\mathcal{A}$-expressions are not available in the surface language for programmers. $\mathcal{A}$-expressions are introduced only by *call/cc* reductions. It turns out that all $\mathcal{A}$-expressions introduced by *call/cc* reductions return the same type as the program itself. That is, in a program:

$$\theta\,!\,E[v\ (\textbf{lambda}\ (x)\ (\mathcal{A}\ E[x]))]$$

that results from a *call/cc* reduction, if the entire program has type τ, then the expression $(\mathcal{A}\ E[x])$ has type τ. Our work on type soundness for a language including **call/cc** shows how to exploit this observation to obtain a proof [64]. The technique adapts without difficulty to a proof of *Type Soundness* for **Imperative Scheme**.

Again, the soft type inference rules for **Imperative Scheme** can be derived from the static type inference rules in the same manner as the soft typing rules for **Pure Scheme** are derived from its static typing rules. Figure 5.4 presents the additional soft typing rules for first-class continuations. Given *Type Soundness*, the proofs for an extended *Correctness* theorem are simple extensions of those for **Box Scheme**.

5.3 Procedure Types of Higher Arity

In **Pure Scheme**, all procedures take exactly one argument. Many programming languages provide procedures that accept different numbers of arguments. Scheme procedures of the form (**lambda** $(x_1 \ldots x_n)$ e) accept n arguments. Scheme procedures of the form (**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 (**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 **lambda**-expression needs a run-time 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 expression (**CHECK-lambda** $(x_1 \ldots x_n)$ e) constructs a procedure that checks to ensure it is passed exactly n arguments. (**CHECK-lambda** $(x_1 \ldots x_n\ .\ x_r)$ e) constructs a procedure that checks to ensure it is passed at least n arguments.

$$\frac{A \models e \Rightarrow e' : (((\tau_1 \rightarrow^{+} \tau_2) \cup \tau_3) \rightarrow^{f} \tau_1) \cup \tilde{\tau}_4 \qquad Empty(\tilde{\tau}_4)}{A \models (\mathbf{call/cc}\ e) \Rightarrow (\mathbf{call/cc}\ e') : \tau_1} \quad (\mathbf{OKcall/cc}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (((\tau_1 \rightarrow^{+} \tau_2) \cup \tau_3) \rightarrow^{f} \tau_1) \cup \tilde{\tau}_4}{A \models (\mathbf{call/cc}\ e) \Rightarrow (\mathbf{CHECK\text{-}call/cc}\ e') : \tau_1} \quad (\mathbf{call/cc}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (((\tau_1 \rightarrow^{+} \tau_2) \cup \tau_3) \rightarrow^{f} \tau_1) \cup \tau_4}{A \models (\mathbf{CHECK\text{-}call/cc}\ e) \Rightarrow (\mathbf{CHECK\text{-}call/cc}\ e') : \tau_1} \quad (\mathbf{CHECK\text{-}call/cc}_{\models})$$

$$\frac{A \models e \Rightarrow e' : \tau_1}{A \models (\mathcal{A}\ e) \Rightarrow (\mathcal{A}\ e') : \tau_2} \quad (\mathcal{A}_{\models})$$

+ {**Box Scheme** soft typing rules (Figure 5.3)}

Figure 5.4 Soft Typing for **Imperative Scheme**

The binary type constructor *(arg · ·)* and the type constant *noarg* encode argument list types. The presentation type *(T1 T2 T3 –> T4)* now abbreviates:

((arg T1 (arg T2 (arg T3 noarg))) –>∗ T4)

where –>∗ 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 *((X1 –> X2) (list X1) –> X2)*, which abbreviates:

((arg ((arg X1 noarg) –>∗ X2) (arg (list X1) noarg)) –>∗ (list X2)).

(Recall that *(list T)* abbreviates a recursive type.) The types of procedures of unlimited arity use recursive types. For example, + sums any number of numbers, and = tests equality for two or more numbers:

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

$$= \; : \; (\textbf{rec}\ ([Y1\ (+\ noarg\ (arg\ num\ Y1))])\ ((arg\ num\ (arg\ num\ Y1)) \longrightarrow\!*\ (+\ false\ true))).$$

Types of primitives that accept optional arguments use unions of *arg* and *noarg*. For example, string–>number accepts a string and an optional base:

string–>number : *((arg str (+ noarg (arg num noarg))) –>* num)*.

When translating to presentation types, procedure types with fixed length argument lists *((arg T1 ... (arg Tn noarg)...) –>* T0)* are abbreviated *(T1 ... Tn –> T0)*. Recursive argument lists are abbreviated *(arglist T)*. Thus the types of map, +, and = are displayed as:

map : *((X1 –> X2) (list X1) –> X2)*
\+ : *((arglist num) –>* num)*
= : *((arg num (arg num (arglist num))) –>* (+ false true))*.

Procedure types representing optional arguments are displayed without abbreviation. While we could design a syntax to abbreviate such procedure types further, they are sufficiently rare that the constructors *arg* and *noarg* are seldom seen by programmers.

Argument list types are similar to ordinary list types. Argument lists use the constructors *arg* and *noarg* where ordinary list types use *cons* and *nil*. Argument lists use different constructors purely for presentation purposes—to distinguish them from ordinary lists.

A consequence of this encoding of procedure types is that it is easy to distinguish (*i*) run-time checks caused by attempting to apply non-procedures (**CHECK-ap**) from (*ii*) run-time checks caused by applying procedures to the wrong number of arguments (**CHECK-lambda**) from (*iii*) other kinds of run-time checks. In practice, we often find that run-time checks of the first two kinds indicate program bugs.

Note. The previous section's treatment of **call/cc** requires minor adaptation when procedures take argument lists. Continuations are represented as procedures that take an argument list of length one. Since continuations require exactly one argument, a continuation may now be applied to the wrong number of arguments. A run-time check to ensure that a continuation is applied to the correct number of arguments belongs in the continuation itself. But continuations are constructed by **call/cc**—there is no **lambda**-expression in the program at which to place a run-time check.

To address this problem, we replace each occurrence of (**call/cc** e) in the program with the expression:

(**call/cc** (**lambda** (k) (e (**lambda** (x) (k x)))))

(taking care to avoid capture of free identifiers in e). This transformation, a composition of two η-expansions, introduces an explicit **lambda**-expression for the continuation. The expression (**lambda** (x) (k x)) will be checked if the continuation may be passed the wrong number of arguments. **End of Note**.

5.4 Letrec

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

(**letrec** ([x_1 e_1] ... [x_n e_n]) e_0)

binds $x_1 \ldots x_n$ to the values of $e_1 \ldots e_n$. The bound identifiers $x_1 \ldots x_n$ are available both in the body e_0 and in the bindings $e_1 \ldots e_n$. The bindings may be arbitrary expressions. They are evaluated in some unspecified order. An expression like (**letrec** ([x (**if** #t x add1)]) (x 1)) that refers to the value of some x_i before e_i has been evaluated is invalid. But R4RS Scheme implementations are not required to detect such invalid expressions. A *conforming* R4RS implementation may instead return an unspecified value for the premature reference to x.

The usual typing rule for **letrec**-expressions in a Hindley-Milner type system is the following:

$$\frac{\begin{array}{c} A[x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_1] \vdash e_1 : \tau_1 \\ \vdots \\ A[x_1 \mapsto \tau_1] \ldots [x_n \mapsto \tau_1] \vdash e_n : \tau_n \\ A[x_1 \mapsto \mathit{Close}(\tau_1, A)] \ldots [x_n \mapsto \mathit{Close}(\tau_n, A)] \vdash e_0 : \tau_0 \end{array}}{A \vdash (\textbf{letrec}\ ([x_1\ e_1]\ \ldots\ [x_n\ e_n])\ e_0) : \tau_0}$$

This typing is not safe with Scheme's semantics for **letrec**-expressions. In the expression (**letrec** ([x (**if** #t x add1)]) (x 1)), this typing rule for **letrec** assigns type *(num -> num)* to x. But as noted above, this expression is invalid. An implementation that does not reject this expression may bind a value outside the type *(num -> num)* to x. In particular, Chez Scheme version 4.1t binds #void to x.

Evaluating this expression under Chez Scheme's optimize-level 3 causes an invalid memory reference as it tries to perform (#void 1).

Several solutions to safely typing **letrec**-expressions are possible. First, if the underlying Scheme implementation guarantees to detect invalid **letrec**-expressions, then the typing above is safe for that particular implementation. Some Scheme interpreters do provide such a guarantee, but the run-time cost of detecting invalid **letrec**-expressions generally precludes compilers from offering it. Second, we could restrict the language to ensure that the bound identifiers $x_1 \ldots x_n$ cannot be used before their values are determined. Standard ML uses this solution, restricting the bindings $e_0 \ldots e_n$ to be **lambda**-expressions. A more general form of this solution allows the bindings to be syntactic values. Third, for an implementation that always returns a unique "unspecified value" like #void, we could include the type of the unspecified value (*void*) in the types of all **letrec**-bound identifiers. A variation of this solution would include the type of the unspecified value in a **letrec**-bound identifier only when the bindings are not syntactic values. Our prototype uses the first solution, which assumes that the underlying Scheme implementation will detect invalid synletrec-expressions.

5.5 Top-level Definitions

A Scheme program is a sequence of definitions whose bodies may refer forwards or backwards to other definitions:

(**define** x_1 e_1)
$\vdots$
(**define** x_n e_n).

An obvious approach to typing programs is to transform each program into a single **letrec**-expression:

(**letrec** ([x_1 e_1]
$\vdots$
[x_n e_n])
#t).

(A Scheme program does not return a useful value, hence the body of the **letrec** is arbitrary.) But treating an entire program as a single **letrec**-expression is not satisfactory because the Hindley-Milner type system assigns polymorphic types to identifiers only within the body of a **letrec**-expression, not within the bindings. That is, according to the typing rule for **letrec**-expressions, the bindings $x_1 \ldots x_n$ have polymorphic types only within the useless **letrec** body, not within $e_1 \ldots e_n$. Were we to treat a Scheme program as a single **letrec**-expression, top-level definitions would not receive polymorphic types.

To obtain polymorphism for top-level definitions, we topologically sort the program's definitions into strongly connected components. The components form a tree that may be organized as small nested **letrec**-expressions and typed in the usual manner. In practice, most components in most Scheme programs include only one element, so this approach is quite effective.

We use the same approach to type internal definitions in **lambda**- and **let**-expressions. This transformation is permitted by the R4RS Scheme semantics since the order in which internal definitions are evaluated is unspecified.

5.6 Pattern Matching

Many modern functional or mostly functional programming languages like 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 like 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 following 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 like:

> (**match** e
> [(#t . #f) e_1]
> [(#f . #t) e_2])

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 [63]. 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.

5.7 Data Definition

Scheme provides eleven 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** (κ $x_1 \ldots x_n$))

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 ... Xn -> (κ X1 ... Xn))*
κ?	:	*(X1 -> (+ true false))*
κ-x_1	:	*((κ X1 ... Xn) -> X1)*
$\vdots$		
κ-x_n	:	*((κ X1 ... Xn) -> Xn)*

set-κ-x_1! : *((κ X1 ... Xn) X1 –> void)*
⋮
set-κ-x_n! : *((κ X1 ... Xn) Xn –> void)*

A **define-structure** definition can be used at top level or as an internal definition within a **lambda**- or **let**-expression. An internal **define-structure** expression is *non-generative*: repeated invocations of its containing **lambda**-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 ('lambda, '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 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 5.5 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 **lambda**- 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 5.1, 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 a facility for defining new immutable forms of data. The definition:

```
(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)
              (Cast 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))
```

Figure 5.5 Expression Parser Type

$$(\textbf{define-const-structure}\ (\kappa\ x_1 \ldots x_n))$$

declares a new type constructor, a value constructor, a predicate, and selectors, just as **define-structure** does. A **define-const-structure** definition does not define any mutators. Applications of a constructor make-κ that was introduced by **define-const-structure** can be treated as syntactic values. Programs that use **define-const-structure** where possible may be assigned more precise types than programs that simply use **define-structure**.

5.8 Records and Modules

Rémy's original work that inspired Cartwright and Fagan's soft type system provides record operations with subtyping. Rémy's record system was motivated by a desire to support object-oriented programming. Our prototype includes record operations with Rémy subtyping in order to support a simple module system.

5.8.1 Records

Let *Field* be a denumerable set of field names. Three new expression forms provide record operations:

$$e \ ::= \ \ldots \ \Big|\ (\textbf{record}\ [a_1\ e_1] \ldots [a_n\ e_n]) \ \Big|\ (\textbf{field}\ a\ e) \ \Big|\ (\textbf{CHECK-field}\ a\ e)$$

where $a \in \mathit{Field}$. (In concrete syntax, field names a and identifiers x are drawn from the same lexical set. They are distinguished by context.) The expression (**record** $[a_1\ e_1] \ldots [a_n\ e_n]$) constructs a record with fields $a_1 \ldots a_n$ bound to the values of $e_1 \ldots e_n$. The expression (**field** $a\ e$) requires e to evaluate to a record and extracts the value of field a from this record. (**CHECK-field** $a\ e$) is similar to (**field** $a\ e$), but checks both that e evaluates to a record and that the record contains field a. Other record operations (test for a record, test for a field, extend a record, modify a field, concatenate records, *etc.* [51]) can also soft typed.

We derive a static type inference system for records by adapting Rémy's work [50, 52]. A new type constructor *record* takes a flag f and an alternative type τ as usual, and a *field list* π:

$$\begin{array}{lcl} \tau & ::= & \ldots \ \Big|\ \mathit{record}^f\ \pi \cup \tau \\ \pi & ::= & a_1^{f_1}\tau_1 * \ldots * a_n^{f_n}\tau_n * \Big(\varpi \ \Big|\ \texttt{\#}\Big) \end{array}$$

A field list consists of zero or more fields $a_i^{f_i}\tau_i$ and either a single field list variable $\varpi \in \mathit{FieldVar}$ or the empty field list **#**. For a field $a_i^{f_i}\tau_i$, the flag f_i indicates whether field a_i is present in the record, and the type τ_i is the type of the field's value.[2] As with union types, field lists must be tidy: a particular field must not be repeated within a field list, and field list variables must have consistent ranges. Hence field lists are formally defined as follows:

$$\pi^W \ ::= \ \varpi^W \ \big| \ \#^W \ \big| \ (a^f\tau^\emptyset)^W * \pi^{W\cup\{a\}} \qquad (a \notin W) \qquad\qquad (\mathit{Field\ Lists})$$

where $W \in \mathbf{2}^{Id}$ is a set of field names). As with union types, the order of fields in a field list does not matter. We define a quotient over field lists to identify different list orders and different representations of the empty field list:

$$\begin{aligned} a_1^{f_1}\tau_1 * a_2^{f_2}\tau_2 * \pi &= a_2^{f_2}\tau_2 * a_1^{f_1}\tau_1 * \pi \\ a^{-}\tau * \# &= \#. \end{aligned}$$

Figure 5.6 presents static type inference rules for record operations that extend the static type system for **Pure Scheme**. The typing rules for records use field flags in a different way than union types use flags. Recall that the union types of value *producers*, like constants and the outputs of a primitive operations, use **+** to indicate "present" types. For fields, the flag **+** still indicates "present", but unchecked **field**-expressions that *consume* values use **+** to demand that the necessary field be present. The inference rule for **record**-expressions allows arbitrary flags for "present" fields so that an expression that chooses one of two records will be typed with the intersection of their fields. For example, the expression:

(**if** e (**record** [x 1][y 2]) (**record**[x 1][z 3]))

has type

$$\begin{aligned} & (\mathit{record}^{+}\ \mathsf{x}^{f_1}\tau * \mathsf{y}^{-}\tau * \mathsf{z}^{-}\tau * \#) \cup \phi \\ = \ & (\mathit{record}^{+}\ \mathsf{x}^{f_1}\tau * \#) \cup \phi \end{aligned}$$

where $\tau = \mathit{num}^{+} \cup \phi$.

Field flags provide a modicum of *record subtyping*. A procedure that takes a record as input may be passed any record that contains at least the required fields, provided

[2]Our system is slightly different from Rémy's. His system places the type of a field with the "present" flag, *i.e.*, flags are **−**, φ, or (**+**τ). Our modification is necessary to accommodate soft typing.

$$\frac{A \vdash e_1 : \tau_1 \qquad \cdots \qquad A \vdash e_n : \tau_n}{A \vdash (\textbf{record}\ [a_1\ e_1] \ldots [a_n\ e_n]) : (\mathit{record}^{\mathbf{+}}\ a_1^{f_1}\tau_1 * \ldots * a_n^{f_n}\tau_n * \mathbf{\#}) \cup \tau} \quad (\textbf{record}_{\vdash})$$

$$\frac{A \vdash e : (\mathit{record}^{f_1}\ a^{\mathbf{+}}\tau_1 * \pi) \cup \emptyset}{A \vdash (\textbf{field}\ a\ e) : \tau_1} \quad (\textbf{field}_{\vdash})$$

$$\frac{A \vdash e : (\mathit{record}^{f_1}\ a^{f_2}\tau_1 * \pi) \cup \tau_2}{A \vdash (\textbf{CHECK-field}\ a\ e) : \tau_1} \quad (\textbf{CHECK-field}_{\vdash})$$

Figure 5.6 Static Typing for Records

that the tail of the field list in the procedure's input type is a polymorphic field list variable. For example, the procedure

```
(define add-a-b
  (lambda (r)
    (+ (field a r) (field b r))))
```

adds the a and b fields of a record together. This procedure has type

$$\forall \varpi\alpha.\, (\mathit{record}^{\mathbf{+}}\ \mathsf{a}^{\mathbf{+}}(\mathit{num}^{\mathbf{+}} \cup \emptyset) * \mathsf{b}^{\mathbf{+}}(\mathit{num}^{\mathbf{+}} \cup \emptyset) * \varpi) \cup \alpha.$$

Because its type is polymorphic in ϖ, this procedure may be applied to a record that has additional fields beyond a and b.

We can adapt this static type system for records to build a soft type system, just as we adapted **Pure Scheme**'s static type system. Figure 5.7 presents the additional soft type inference rules for records. Recall that there is a distinguished subset of *absent* flag variables that are not generalized by *SoftClose* and that induce a subset of *absent* flags $\tilde{f} \in \{f \mid FV(f) \subset \mathit{AbsFlagVar}\}$. **Pure Scheme**'s soft type system uses absent flag and type variables in place of **–** and $\emptyset$ in the input types of primitive operations for two reasons. First, absent variables ensure that all applications of primitive operations can be typed. Second, absent variables indicate where to insert run-time checks. For records, we need absent variables only for the second reason. The soft type inference rule for **field**-expressions uses an absent flag in place of **+** for

$$\frac{A \models e_1 \Rightarrow e_1' : \tau_1 \qquad \cdots \qquad A \models e_n \Rightarrow e_n' : \tau_n}{\begin{array}{l} A \models (\textbf{record}\ [a_1\ e_1] \ldots [a_n\ e_n]) \Rightarrow (\textbf{record}\ [a_1\ e_1'] \ldots [a_n\ e_n']) \\ \quad : (\mathit{record}^{\textbf{+}}\ a_1^{f_1} \tau_1 * \ldots * a_n^{f_n} \tau_n * \textbf{\#}) \cup \tau \end{array}} \quad (\textbf{record}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (\mathit{record}^{f_1}\ a^{\tilde{f}_2} \tau_1 * \pi) \cup \tilde{\tau}_2 \qquad \mathit{Empty}\{\tilde{\tau}_2\} \qquad \tilde{f}_2 \neq \textbf{-}}{A \models (\textbf{field}\ a\ e) \Rightarrow (\textbf{field}\ a\ e') : \tau_1} \quad (\textbf{OKfield}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (\mathit{record}^{f_1}\ a^{\tilde{f}_2} \tau_1 * \pi) \cup \tilde{\tau}_2}{A \models (\textbf{field}\ a\ e) \Rightarrow (\textbf{CHECK-field}\ a\ e') : \tau_1} \quad (\textbf{field}_{\models})$$

$$\frac{A \models e \Rightarrow e' : (\mathit{record}^{f_1}\ a^{f_2} \tau_1 * \pi) \cup \tau_2}{A \models (\textbf{CHECK-field}\ a\ e) \Rightarrow (\textbf{CHECK-field}\ a\ e') : \tau_1} \quad (\textbf{CHECK-field}_{\models})$$

Figure 5.7 Soft Typing for Records

the extracted field. If this flag is instantiated to **-**, the **field**-expression requires a run-time check.

5.8.2 Modules

Our prototype provides a simple module facility via two expression forms:

$$\begin{array}{rcl|l} e & ::= & \ldots & (\textbf{module}\ (x_1 \ldots x_n)\ \mathit{definitions}) \\ & & & (\textbf{import}\ ([e_1\ x_{11} \ldots x_{1m_1}] \ldots [e_n\ x_{n1} \ldots x_{nm_n}])\ e) \end{array}$$

A **module**-expression constructs a module that exports names $x_1 \ldots x_n$. The values of these names are taken from the values of the lexical identifiers $x_1 \ldots x_n$ that are visible at the end of the **module**-expression's body. Our intention is that the exported names be bound by internal definitions within the module, as the following example illustrates:

```
(define M
  (module (make-set union element-of?)
    (define make-set list)
    (define union append)
    (define element-of? member)))
```

An **import**-expression extracts exported names from one or more modules, binding them to the same-named lexical identifiers (our prototype's **import**-expression can also rename imported bindings). The expressions $e_1 \ldots e_n$ must evaluate to modules. The names $x_{i1} \ldots x_{im_i}$ are imported from module e_i and bound in e.

Our modules are merely syntactic sugar for records. A **module**-expression expands to a **record**-construction wrapped by a data constructor:

```
(let ()
  definitions
  (make-module (record (x1 x1) ... (xn xn))))
```

An **import**-expression expands to **field**-expressions:

```
(let ([m1 (module-internals e1)]
      ...
      [mn (module-internals en)])
  (let ([x11 (field x11 m1)] ... [x1m1 (field x1m1 m1)]
        ...
        [xn1 (field xn1 mn)] ... [xnm1 (field xnm1 mn)])
  e))
```

where $m_1 \ldots m_n$ are fresh identifiers not free in e. The procedures make-module and module-internals are defined by a single global data definition:

```
(define-const-structure (module internals))
```

The procedure module-internals is accessible only to the **import** macro.

An **import**-expression may require run-time checks to ensure that the imported identifiers are actually exported by the corresponding modules. For example, in the following program:

```
(define m1 (module (a) (define a 1)))
(define m2 (module (b) (define b #t)))
(define m3 (if P m1 m2))
(import ([m3 a])
  (add1 a))
```

the name a is available to import from module m3 only if P yields true. Since module operations expand into record operations, **import**-expressions that require run-time checks will result in appropriate **CHECK-field** operations. The **import**-expression in the above program expands into:

```
(let ([m-fresh (module-internals m3)])
  (let ([a (CHECK-field a m-fresh)]) (add1 a)))
```

5.9 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:

$$e \ ::= \ \ldots \ \Big| \ (\mathbf{:} \ \tau \ e)$$

with the following type inference rule:

$$\frac{A \vdash e : \tau}{A \vdash (\mathbf{:} \ \tau \ e) : \tau}$$

The type τ must have no free type variables. Semantically, an explicit type annotation behaves as the identity function. The typing rule rejects programs for which the subexpression e does not have the specified 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. 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 type annotation like (: *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 like (: *(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 3.8). Any program that reaches an unsatisfied annotation will terminate with an error message.

5.10 Macro Definitions

Syntactic abstractions or *macro definitions* are an important tool for eliminating programming patterns that procedural abstraction cannot address. For example, macros are essential to our definitions of **module**- and **import**-expressions since they are new binding forms.

Our prototype accommodates macro definitions by expanding invocations of user-defined macros before type checking. The prototype also expands some core expression forms like **cond**- and **do**-expressions, but not forms key to soft typing like **let**-, **letrec**-, and **match**-expressions.

Unfortunately, expanding macros means that types are reported for the expanded program rather than the original source code. The programmer must reason about the expanded program and carry this reasoning back to the source code. Automatically transforming type information about an expanded program back to the unexpanded form is a difficult problem that is beyond the scope of this work. Fortunately, user-defined macros are relatively rare. We have encountered no difficulties in manually translating type information back to source code.

Chapter 6

Related Work

In this chapter we discuss other work related to soft typing. We begin with work directly concerning soft typing. In particular, we discuss Cartwright and Fagan's soft type system, on which our work is based. We discuss systems for optimizing run-time checking and tagging in dynamically typed languages, and we comment on the relation of static type systems to soft typing.

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 for an idealized functional language [10, 16]. 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 developed to reduce record subtyping to polymorphism [50]. 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 like

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

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

$$(\mathit{false}^{+} \cup \mathit{num}^{\varphi_2} \cup \ldots \cup (\mathit{cons}^{\varphi_{n-1}}\ \alpha_{n-1}\ \alpha'_{n-1}) \cup (\alpha_n \rightarrow^{\varphi_n} \alpha'_n)) \rightarrow^{+}$$
$$(\mathit{false}^{-} \cup \mathit{num}^{\varphi_2} \cup \ldots \cup (\mathit{cons}^{\varphi_{n-1}}\ \alpha_{n-1}\ \alpha'_{n-1}) \cup (\alpha_n \rightarrow^{\varphi_n} \alpha'_n)).$$

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.

(i) Our system presents types to programmers in a simple presentation type language that is easy to interpret. We argued the importance of presentation types in Chapter 4.

(ii) Our system addresses various features of realistic programming languages (see Chapter 5), like assignment, first-class continuations, pattern matching, data definition, etc. Cartwright and Fagan's system deals with only a functional language with a simple case statement.

(iii) 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 like assignment and continuations (see Chapter 3 and Appendix A). At present, we are not aware of any established techniques based on denotational semantics for proving type soundness for imperative languages.

(iv) 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. We present our conclusions in Chapter 7.

6.1.2 Aiken, Wimmers, and Lakshman

Aiken, Wimmers, and Lakshman recently developed a sophisticated soft type system for the functional language FL [3, 4]. 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 Static Typing 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 system for optimizing run-time tagging and checking [26] inserts only atomic coercions like $num \rightsquigarrow \top$ and $\top \rightsquigarrow num$ that are small, constant time operations. Systems proposed by Gomard [21] and Thatte [58, 59] and studied by

O'Keefe and Wand [46], as well as a more general system proposed by Henglein [25], insert compound coercions like *(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 $\top \to \top$. 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. Hence we do not consider them to be soft type systems. 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 Flow Analysis

The designers of optimizing compilers for Scheme and Lisp have developed type recovery procedures based on iterative flow analysis [8, 33, 39, 40]. Some object-oriented languages use similar methods [12, 34]. 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; 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 developed a family of techniques based on abstract interpretation to perform flow analysis for Scheme-like languages [55]. 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 [55: p. 121]. Hence this algorithm is likely to be impractical for large programs that make extensive use of higher-order functions.

Heintze has recently developed a method of performing flow analysis for higher-order languages based on ignoring dependencies between identifiers [24]. He reports reasonable execution times to analyze ML programs of several thousand lines. As the information his system infers is fairly precise, we believe that his analysis could be used to perform type recovery. To our knowledge, this application of his analysis has not been investigated.

Jagannathan and Weeks [31] have recently implemented an extension of Shivers's 0CFA technique. Their new algorithm appears to yield type information of comparable precision to that of Heintze's system with reasonable execution times.

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 like object-oriented programming, and nourishes a never-ending search for better static type systems. Consequently, realistic statically typed languages like 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 like Standard ML [44] and Modula 3 [9] have sophisticated type systems that include polymorphism and subtyping to achieve greater flexibility, but even implementations of these languages invariably include loopholes, like the procedure `System.Unsafe.cast` in Standard ML of New Jersey.

Several authors [1, 2, 37] 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 like persistent storage. These extensions do not alter the fundamental character of static type systems.

Freeman and Pfenning [20, 19] 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, run-time checks do not alter the fundamental character of static type systems.

Chapter 7

Experiences with a Prototype Soft Type System

Soft Scheme is 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. In this chapter, we illustrate Soft Scheme with several examples. We discuss the utility of the type information inferred by Soft Scheme. We show how soft typing compares in practice to static and dynamic typing and how Soft Scheme may be used to engineer prototypes into robust and efficient programs. some problems we have encountered in using our prototype.

7.1 Soft Scheme

Soft Scheme performs batch type checking for complete R4RS Scheme [13] programs. When applied to a program, the type checker writes a version of the program containing explicit run-time checks to an output file and displays only a summary of the inserted run-time checks. Programmers may then inspect type information interactively according to their interest. The output program can safely be executed by a Scheme compiler that disables all run-time checks.[1]

Our first example illustrates the operation of Soft Scheme. The following program defines and uses a function flatten that flattens an arbitrarily nested list of lists (a tree) to a proper list of leaves:

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

[1]Provided that the compiler treats **letrec**-expressions as Section 5.4 indicates.

This program binds b to the list '(1 2 3).

For the above program, Soft Scheme displays the following summary of run-time checks:

```
TOTAL CHECKS    0   (of 10 is 100.0%)
```

This program requires no run-time checks as it is typable in our underlying static type system. The summary also indicates the number of potential sites for run-time checks. Interactive commands permit inspecting the types of the top-level definitions:

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

The type of a reflects the shape of the value '(1 (2) 3), which abbreviates the value (cons 1 (cons (cons 2 '()) (cons 3 '()))). Pairs, which are constructed by cons, have type *(cons · ·)*. The empty list '() has type *nil*. The type for b indicates that b is a proper list of numbers. The type *(list num)* abbreviates

(rec ([Y (+ nil (cons num Y))]) Y),

which denotes the least fixed point of the recursion equation

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

Finally, flatten's type *(Y1 -> (list ...))* indicates that flatten is a procedure of one argument returning a list. The argument type is defined by

$$Y1 = nil \cup (cons\ Y1\ Y1) \cup X1,$$

where *X1* is a type variable. Hence, flatten accepts the empty list, pairs, or any other kind of value, *i.e.*, flatten accepts any value. The result type of flatten is *(list (+ (not nil) (not cons) X1))*. A result returned by flatten is a proper list of elements of type *X1*, that does not include pairs or the empty list.

Now suppose we add the following lines to our program:

```
(define c (car b))
(define d (map add1 a))
(define e (map sub1 (flatten '(this (that)))))
(define f (flatten a b))
(define g ((if (read) flatten #f) b))
```

Type checking the extended program yields the summary:

flatten	1	(1 lambda)
c	1	(1 prim)
d	1	(1 prim)
e	1	(1 prim) (1 ERROR)
g	1	(1 ap)
TOTAL CHECKS	5	(of 18 is 27.0%) (1 ERROR)

This extended program requires five run-time checks. Three run-time checks are required at primitive operations, one in each of the definitions of c, d, and e. One run-time check for the correct number of arguments is required at the **lambda**-expression in flatten. One run-time check for a procedure in the function position is required at the outermost application in g. The program output by Soft Scheme shows the locations of the run-time checks:

```
(define flatten
  (CHECK-lambda (l)
    (cond [(null? l) '()]
          [(pair? l) (append (flatten (car l)) (flatten (cdr l)))]
          [else (list l)])))
(define a '(1 (2) 3))
(define b (flatten a))
(define c (CHECK-car b))
(define d (map CHECK-add1 a))
(define e (map ERROR-sub1 (flatten '(this (that)))))
(define f (flatten a b))
(define g (CHECK-ap (if (read) flatten #f) b))
```

The prefixes **CHECK-** and **ERROR-** indicate run-time checks.[2] An unnecessary check is inserted at car because b's type

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

includes *nil*, which is not a valid input to car. When executed, this run-time succeeds. **CHECK-**add1 indicates that add1 may fail when applied to some element of a, as

[2]We have simplified the output program to clarify our presentation. The program produced by Soft Scheme includes an identifying number with each run-time check, which may be used to inspect the type of that primitive operation. Macros like **cond** are expanded, and some **if**-expressions are transformed to **match**-expressions as discussed in Section 5.6.

indeed it does when applied to '(2), the second element of a. **ERROR-**sub1 indicates that the occurrence of sub1 in this program never succeeds—if it is ever reached, it will fail. **CHECK-lambda** in the definition of flatten indicates that this **lambda**-expression must check that it receives the correct number of arguments. Most calls to flatten in this program pass the correct number of arguments (one), but the call in f will pass the wrong number of arguments if it is reached. Finally, **CHECK-ap** indicates that the function position (**if** (read) flatten #f) of the application in g must be tested to ensure that it evaluates to a procedure. No other run-time checks are required to ensure safe execution of this program. In particular, no run-time checks are required in the bodies of the library routines map and append.

This example illustrates three kinds of run-time checks: primitive checks, application checks, and argument count checks. Any of these kinds of checks may be classified as errors if the type checker can prove that the check never succeeds. Soft Scheme also inserts run-time checks at **match**-expressions where the type of the matched value exceeds the type produced by folding the pattern types together. For example, the program

```
(match (read)
  [(a . ()) a])
```

receives a run-time check as follows:

```
(CHECK-match (read)
  [(a . ()) a]).
```

This run-time check is required because the **match**-expression requires the matched value be a proper list of length one, but Scheme's read primitive can return any s-expression. Inexhaustive matches are another cause of **match**-expression run-time checks (see Section 5.6).

To guarantee that enough run-time checks are inserted to ensure the safety of the output program, Scheme programs input to our type checker must not have free identifiers. Nevertheless, to allow typing partial programs, Soft Scheme assumes the liberal type $\forall\alpha.\alpha$ for any missing definitions. The type checker issues warnings about such missing definitions because a program with missing definitions is not guaranteed type safe.

7.2 The Utility of Type Information

Our next example illustrates the utility of type information for reasoning about Scheme programs and for finding bugs.

Our example concerns *boolean formulae* that have the representation

$$b \;::=\; \texttt{\#t} \;\big|\; \texttt{\#f} \;\big|\; (\textbf{lambda}\ (x)\ b).$$

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
$\neg x$	: (**lambda** (x) (**not** x))
$x \wedge y$	: (**lambda** (x) (**lambda** (y) (**and** x y)))
$x \vee (y \wedge z)$	: (**lambda** (x) (**lambda** (y) (**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
  (lambda (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 (lambda (x) (lambda (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. It is usually the types of a program's top-level definitions rather than run-time checks that indicate program bugs. For example, consider the following incorrect variation of the above program:

```
(define wrong-taut
  (lambda (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 (lambda (x) (lambda (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 *(+ true false)*, not just *true*. Applied to (**lambda** (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 ...)                                              (7.1)
(define d (taut taut))
```

Soft Scheme inserts no run-time checks for this program and assigns **d** the type *(+ true false)*. 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

(Y1 –> (+ true false)) $\subseteq$ *Y1*.

Expanding *Y1* we obtain

(Y1 –> (+ true false)) $\subseteq$ *(+ true false ((+ true false) –> Y1))*,

which simplifies to

(Y1 –> (+ true false)) $\subseteq$ *((+ true false) –> Y1)*.

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

(+ true false) $\subseteq$ *Y1*

which certainly holds.

Note. The application (**taut taut**) illustrates the importance of our type swapping extension for obtaining more precise types (Section 3.7.2). Without this extension, Soft Scheme inserts three run-time checks into program 7.1 as follows:

```
(define taut
  (lambda (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 fails to provide the subtyping containment *(Y1 -> (+ true false))* $\subseteq$ *Y1*. From the definition of `taut` alone, the internal type inferred for `taut` is[3]

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

Typing the application (`taut taut`) requires replacing $\tilde{\alpha}_4$ with $\mathit{true}^{+} \cup \mathit{false}^{+} \cup \alpha_5$. 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

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

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

7.3 Overcoming the Limitations of Static Typing

The tautology checker of the previous section illustrates a limitation of statically typed languages that our soft type system overcomes. Our representation of boolean formulae is convenient for reusing existing procedures like `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 procedure 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:

---

[3]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`.

```
(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
  (lambda (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 (lambda (x) (C (not x))))))
(define c (ml-taut (O (lambda (x) (O (lambda (y) (C (and x y))))))))
(define d (ml-taut (O (lambda (x) (C (ml-taut (C x)))))))
```

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

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

Figure 7.1 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 formula. Hence an ML programmer would most likely choose a different representation for boolean formulae than ours.

Cartwright and Felleisen [11] present a more complex example that illustrates how our soft type system overcomes the limitations of traditional static type systems.

```
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)))))
```

Figure 7.1 Standard ML Version of `taut`

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 [56] attempts to use a similar approach in Haskell [30] to compose interpreters from pseudo-monads. 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.

7.4 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 measurement and 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. We can measure the speed at which our prototype analyses typical programs.

7.4.1 Minimizing Run-time Checking

Figure 7.2 summarizes run-time checking for the Scheme versions of the Gabriel Common Lisp bench marks[4] and two other programs. The extra two programs, *Dtype* and *Interp*, are Henglein's dynamic type inference prototype applied to itself [26] and Cartwright and Felleisen's extensible denotational framework [11]. The former is representative of a large Scheme program that has not been engineered with soft typing in mind. The latter is a large Scheme program that was specifically designed using Soft Scheme. The percentages indicate how frequently our system inserts run-time checks compared to conventional dynamic typing. The static frequency indicates the incidence of run-time checks inserted in the source code. The dynamic frequency indicates how often the inserted checks are executed. For example, Soft Scheme places run-time checks at 10% of the potential sites for run-time checks in the *Boyer* bench mark. Only 5% of the potential sites *Boyer* encounters during execution involve run-time checks.

Some programs like *Browse*, *Div*, and *Fft* 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 will examine *Fft* in detail in the next section.

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

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

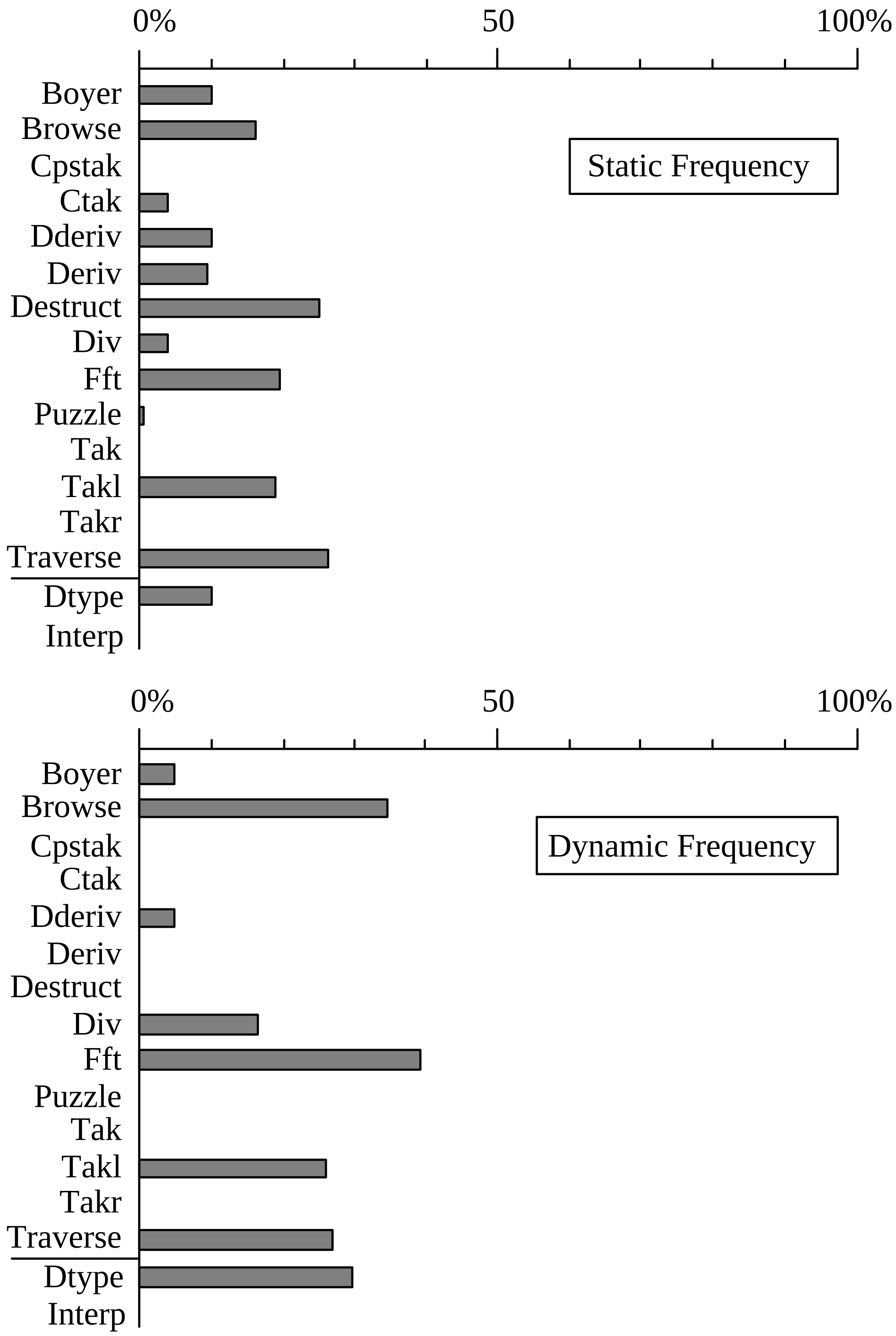

Figure 7.2 Reduction in Run-Time Checking

7.4.2 Execution Time

Figure 7.3 indicates the break down of execution time for the Gabriel bench marks under Chez Scheme 4.1t on an unloaded SparcStation 1 with 64 megabytes of physical memory. 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 bench marks 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 naïve compiler. The grey section indicates the time each bench mark spends performing run-time checks under soft typing. This fraction was measured by using optimize-level 3 to turn off all run-time checks other than those explicitly inserted by Soft Scheme.[5] Thus the white section indicates the execution time soft typing saves by eliminating unnecessary run-time checks.

Soft Scheme significantly decreases time spent performing run-time checks for most of the bench marks, even though these programs were not written with soft typing in mind. Whether this reduction leads to a significant performance improvement depends on how often the code containing the eliminated checks is executed. Programs like *Browse* and *Traverse*, for example, do not expose enough type information to enable the type checker to remove critical run-time checks. Programs like *Cpstak*, *Ctak*, *Destruct*, and *Tak* spend no appreciable fraction of time performing run-time checks. But *Puzzle* illustrates the dramatic benefit (speedup by a factor of 3.3) that can be obtained when run-time type checks are removed from inner loops.

7.4.3 Speed of Analysis

Although our prototype is not a highly tuned program, analysis times for moderate size programs are usually reasonable. Most of the Gabriel bench marks are checked by Soft Scheme in under a second. The space requirements of our prototype are also

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

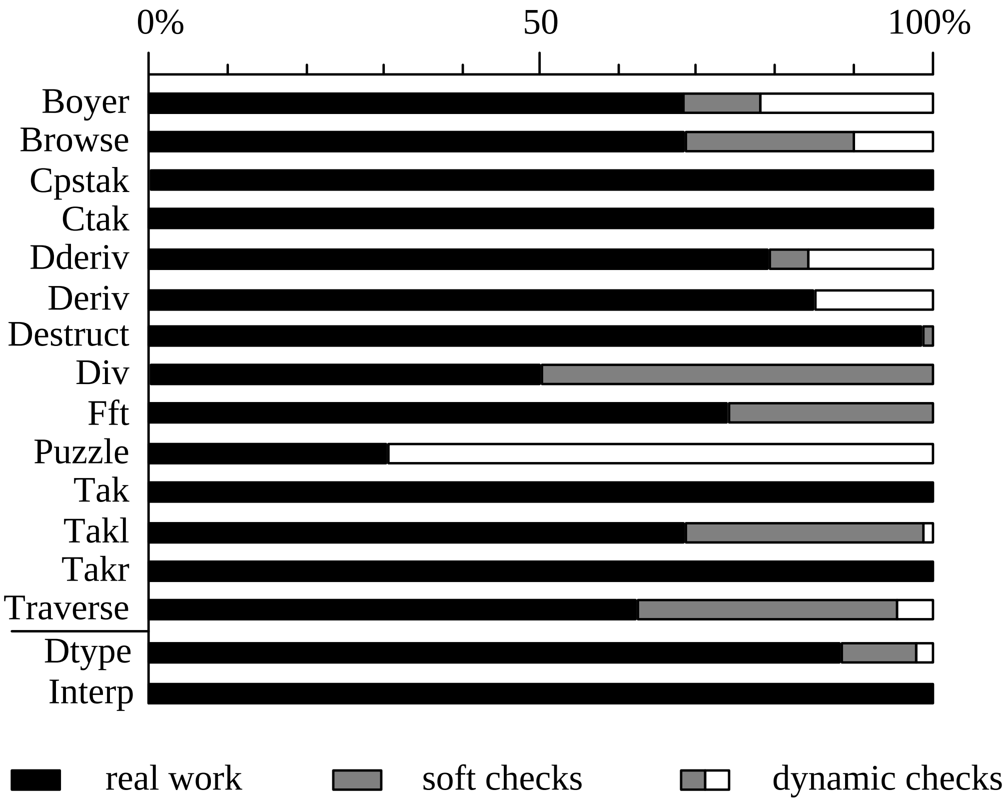

Figure 7.3 Break Down of Execution Times

modest, and generally correlate with analysis times. Figure 7.4 presents the time and space requirements of our prototype for several of the larger bench marks. These times were collected under Chez Scheme 4.1b on an unloaded SparcStation 10.

Soft type checking involves inferring types and inserting run-time checks. Inserting run-time checks typically consumes only a small fraction of total analysis time. Our prototype spends the majority of its time inferring types. The primary concern in making type inference run fast is minimizing the space requirements of types. As our current prototype uses several suboptimal algorithms, we expect that the performance of our system will improve in the future.

7.5 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 efficient production programs. We take the four Gabriel bench marks that exhibit the greatest potential for improvement from Figure 7.3. Guided by the information produced by Soft Scheme, we apply simple semantics-preserving transformations to rewrite *Fft*, *Div*, *Takl*, and *Traverse* to run significantly faster.

7.5.1 Fft

Examining the source to *Fft*, we find the following code fragment:

```
(define (fft areal aimag)
  (let ([ar 0]
        [ai 0])
    (set! ar areal)
    (set! ai aimag)
      ⋮
    many uses of ar, ai via vector-ref, vector-set!
```

The arguments areal and aimag are always vectors. Except for their initial useless numeric values, ar and ai are always treated as vectors; they are never reassigned. But their initial values cause the type inferred for both ar and ai to be *(+ num (vec num))*. Hence all uses of ar and ai as vectors require run-time checks. By simplifying the code as follows:

	Program Size (Kbytes)	*Analysis Time* (sec)	*Analysis Space* (Mbytes)
Boyer	21.4	8.3	13.8
Browse	8.3	1.5	2.2
Takr	17.5	3.8	6.5
Traverse	5.2	1.1	1.6
Dtype	69.0	17.7	22.7
Interp	8.6	1.5	3.3

Figure 7.4 Time and Space Requirements

```
(define (fft areal aimag)
  (let ([ar areal]
        [ai aimag])
     ⋮
    many uses of ar, ai via vector-ref, vector-set!
```

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

7.5.2 Div

Div contains a procedure that constructs lists of length n:[6]

```
(define create-n
  (lambda (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. But the program only calls create-n with an even number, and other procedures assume that the lists con-

[6] We have taken the liberty of rewriting this procedure to avoid Scheme's **do**-construct.

structured are of even length. Since the result type of create-n includes odd length lists, these procedures require run-time checks. If we unroll the loop in create-n:

```
(define create-n-even
  (lambda (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. If we replace create-n with create-n-even, the procedures that assume their inputs are even length lists now require no run-time checks.

7.5.3 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.

By rewriting shorterp in a different style:

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

Soft Scheme infers a more precise type for shorterp:

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

This transformation eliminates two of the five run-time checks from *Tak*.

7.5.4 Traverse

Figures 7.5a, 7.5b, 7.5c, 7.5d, and 7.5e present the original Gabriel bench mark *Traverse*.[7] This program constructs a large graph of *nodes* and then traverses the graph, changing the values of various fields in nodes. Figure 7.5b defines procedures to construct and access nodes. These definitions represent nodes as 11-element vectors. Figure 7.5c defines several procedures to manipulate special *node-lists*. Figure 7.5d defines the procedures that create the graph of nodes. And Figure 7.5e defines the procedures that traverse the graph. The bench mark is run by invoking:

```
(init-traverse)
(run-traverse)
```

As a dynamically typed program, this bench mark typically runs in about 33.7 seconds on an unloaded SparcStation 1 under Chez Scheme 4.1t with maximum safe optimization (optimize-level 2).

Soft Scheme inserts 51 run-time checks in this original version of *Traverse*. The soft typed version typically shows little if any speedup over the original dynamically typed program. Soft Scheme is unable to infer sufficiently precise types for this program to eliminate important run-time checks. Although the nodes manipulated by create-structure are built from 11-element vectors, create-structure has type:

(**rec** *((Y1 (+ num nil (vec Y2) (cons Y1 Y1)))*
(Y2 (+ num nil sym (cons Y1 Y2) false true)))
(num -> Y1)))

This type does not even reflect the length of the vectors representing nodes.

By using our data definition facility, we can improve the types inferred for nodes. We replace the definitions in Figure 7.5b with Figure 7.6. Soft Scheme inserts 51 run-time checks in the modified program. (Coincidentally, the same number of checks are inserted as before, but at different places.) The procedure create-structure now has type:

[7] To clarify the presentation, we have taken the liberty of expanding **do**-expressions and quasi-quotes, renamed a few identifiers, and performed some trivial reformatting.

```
(define *sn* 0)
(define *rand* 21)
(define *count* 0)
(define *marker* #f)
(define *root* '())

(define (snb) (set! *sn* (+ 1 *sn*)) *sn*)

(define (random) (set! *rand* (remainder (* *rand* 17) 251)) *rand*)
```

Figure 7.5a Miscellaneous Initialization for *Traverse*

```
(rec ((Y1 (+ num
             nil
             (cons Y1 Y1)
             (node (list Y1) (list Y1) num nil nil nil nil nil nil nil))))
  (num -> Y1))
```

A node's type now correctly represents the number of fields in a node, and even gets the types of the fields about right. This program typically runs in 31.4 seconds, which represents an improvement of about 7%.

The problem is that the type of `create-structure` is still quite imprecise. This procedure returns only nodes. It never returns a number, nil, or pair. In Figure 7.5d, we see that identifiers `x` and `y` are assigned nodes but initialized to the number `0`. Also the identifier `a` is used in two distinct ways. It is first used to build a loop of nodes in `p`, and then reused as a node-list object by the code `(set! a (cons p '()))`. By rewriting `create-structure` to avoid these peculiarities, we can eliminate *num*, *nil*, and *cons* from the type of a node returned by `create-structure`. Figure 7.7 presents the new version of `create-structure`. In preparation for our next transformation, we have also changed `create-structure` to call the node-list operations rather than embed knowledge of the representation of node-lists. The procedure `traverse-empty?` is a new function that tests whether a node-list is empty.

```
(define (mk-node)
  (let ([node (make-vector 11 '())])
    (vector-set! node 0 'node)
    (vector-set! node 3 (snb))
    node))


(define (node-parents node) (vector-ref node 1))
(define (node-sons node) (vector-ref node 2))
(define (node-sn node) (vector-ref node 3))
(define (node-entry1 node) (vector-ref node 4))
(define (node-entry2 node) (vector-ref node 5))
(define (node-entry3 node) (vector-ref node 6))
(define (node-entry4 node) (vector-ref node 7))
(define (node-entry5 node) (vector-ref node 8))
(define (node-entry6 node) (vector-ref node 9))
(define (node-mark node) (vector-ref node 10))


(define (set-node-parents! node v) (vector-set! node 1 v))
(define (set-node-sons! node v) (vector-set! node 2 v))
(define (set-node-sn! node v) (vector-set! node 3 v))
(define (set-node-entry1! node v) (vector-set! node 4 v))
(define (set-node-entry2! node v) (vector-set! node 5 v))
(define (set-node-entry3! node v) (vector-set! node 6 v))
(define (set-node-entry4! node v) (vector-set! node 7 v))
(define (set-node-entry5! node v) (vector-set! node 8 v))
(define (set-node-entry6! node v) (vector-set! node 9 v))
(define (set-node-mark! node v) (vector-set! node 10 v))
```

Figure 7.5b Original *Traverse* Node Definitions

```
(define (traverse-remove n q)
  (cond [(eq? (cdr (car q)) (car q))
         (let ([x (caar q)])
           (set-car! q '())
           x)]
        [(zero? n)
         (let ([x (caar q)])
           (let loop ([p (car q)])
             (if (eq? (cdr p) (car q))
                 (begin
                   (set-cdr! p (cdr (car q)))
                   (set-car! q p))
                 (loop (cdr p))))
           x)]
        [else (let loop ([n n] [q (car q)] [p (cdr (car q))])
                (if (zero? n)
                    (let ([x (car q)]) (set-cdr! q p) x)
                    (loop (- n 1) (cdr q) (cdr p))))]))


(define (traverse-select n q)
  (let loop ([n n] [q (car q)])
    (if (zero? n) (car q) (loop (- n 1) (cdr q)))))


(define (traverse-add a q)
  (cond [(null? q)
         (let ([x (cons a '())])
           (set-cdr! x x)
           (cons x '()))]
        [(null? (car q))
         (let ([x (cons a '())])
           (set-cdr! x x)
           (set-car! q x)
           q)]
        [else (set-cdr! (car q) (cons a (cdr (car q)))) q]))
```

Figure 7.5c Original *Traverse* Node-list Operations

```
(define (create-structure n)
  (let ([a (cons (mk-node) '())])
    (let loop ([m (- n 1)] [p a])
      (if (zero? m)
          (begin
            (set-cdr! p a)
            (set! a (cons p '()))
            (let loop2 ([unused a]
                        [used (traverse-add (traverse-remove 0 a) '())]
                        [x 0]
                        [y 0])
              (if (null? (car unused))
                  (find-root (traverse-select 0 used) n)
                  (begin
                    (set! x (traverse-remove
                              (remainder (random) n)
                              unused))
                    (set! y (traverse-select
                              (remainder (random) n)
                              used))
                    (traverse-add x used)
                    (set-node-sons! y (cons x (node-sons y)))
                    (set-node-parents! x (cons y (node-parents x)))
                    (loop2 unused used x y)))))
          (begin
            (set! a (cons (mk-node) a))
            (loop (- m 1) p))))))


(define (find-root node n)
  (let loop ([n n])
    (if (or (zero? n) (null? (node-parents node)))
        node
        (begin
          (set! node (car (node-parents node)))
          (loop (- n 1))))))
```

Figure 7.5d Original *Traverse* Traversal Routine

```
(define (travers node mark)
  (if (not (eq? (node-mark node) mark))
      (begin
        (set-node-mark! node mark)
        (set! *count* (+ 1 *count*))
        (set-node-entry1! node (not (node-entry1 node)))
        (set-node-entry2! node (not (node-entry2 node)))
        (set-node-entry3! node (not (node-entry3 node)))
        (set-node-entry4! node (not (node-entry4 node)))
        (set-node-entry5! node (not (node-entry5 node)))
        (set-node-entry6! node (not (node-entry6 node)))
        (let loop ([sons (node-sons node)])
          (if (not (null? sons))
              (begin
                (travers (car sons) mark)
                (loop (cdr sons))))))))


(define (traverse root)
  (let ([*count* 0])
    (travers root (begin (set! *marker* (not *marker*)) *marker*))
    *count*))


(define (init-traverse)
  (set! *root* (create-structure 100)))
  #f)


(define (run-traverse)
  (let loop ([i 50])
    (if (not (zero? i))
        (begin
          (traverse *root*)
          (traverse *root*)
          (traverse *root*)
          (traverse *root*)
          (traverse *root*)
          (loop (- i 1))))))
```

Figure 7.5e Original *Traverse* Driver Code

```
(define-structure
  (node parents sons sn entry1 entry2 entry3 entry4 entry5 entry6 mark))
(define mk-node
  (lambda ()
    (make-node '() '() (snb) '() '() '() '() '() '() '())))
```

Figure 7.6 Improved Node Definitions for *Traverse*

```
(define traverse-empty?
  (lambda (q)
    (null? (car q))))


(define (create-structure n)
  (let ([a (traverse-add (mk-node) '())])
    (let loop ([m (- n 1)])
      (if (zero? m)
          (let loop2 ([unused a]
                      [used (traverse-add (traverse-remove 0 a) '())])
            (if (traverse-empty? unused)
                (find-root (traverse-select 0 used) n)
                (let* ([x (traverse-remove
                            (remainder (random) n)
                            unused)]
                       [y (traverse-select
                            (remainder (random) n)
                            used)])
                  (traverse-add x used)
                  (set-node-sons! y (cons x (node-sons y)))
                  (set-node-parents! x (cons y (node-parents x)))
                  (loop2 unused used))))
          (begin
            (traverse-add (mk-node) a)
            (loop (- m 1)))))))
```

Figure 7.7 Improved Traversal Routine for *Traverse*

With the improved version of create-structure in Figure 7.7, Soft Scheme inserts 40 run-time checks. The type of create-structure is much more precise:

*(***rec** *((Y1 (node (list Y1) (list Y1) num nil nil nil nil nil nil nil)))*
(num -> Y1))

But the program shows no significant improvement in execution time from the previous version. Of the 40 remaining run-time checks, 18 appear in the node-list operations in Figure 7.5c. Most of these run-time checks appear at car and cdr operations. These operations represent a non-empty node-list as pair whose first component is a circular chain of pairs. The first elements of the pairs in this chain are the elements of the node-list. An empty node-list is a pair whose first component is '(). Hence the type inferred for a node-list is *(cons (list (node ...)) nil)*. Operations manipulating the chain of a non-empty list require run-time checks because the chain type *(list (node ...))* includes *nil*.

By changing the representation of an empty node-list, we can eliminate *nil* from the chain type. Our new representation for node-lists, shown in Figure 7.8, represents an empty node-list as a pair whose *second* component is #f. Non-empty node-lists have second component #t. To create an initial node-list, we assume a version of **letrec** that permits the definition of recursive data. With this new representation for node-lists, a non-empty node list has type:

*(***rec** *([Y1 (cons (node ...))]) (cons Y1 true))*.

This transformation eliminates the run-time checks in the node-list operations, reducing the total number of run-time checks to 22. The program now runs in 29.9 seconds.

The remaining run-time checks are mainly in the driver routines in Figure 7.5e. Observe that these routines assign nodes to identifier *root*, but *root* is initialized to '() in Figure 7.5b. Hence *root* has type *(+ nil (node ...))*. The presence of *nil* in this type forces run-time checks at operations that manipulate nodes, particularly those in travers. By rewriting the driver code to avoid the initial value '() as in Figure 7.9, we can eliminate the run-time checks at node manipulating primitives. In the final version of the program, there is a single run-time check at car in find-root. The final version of the bench mark runs in 21.6 seconds, which consumes about 33%

```
(define (traverse-remove n q)
  (cond [(eq? (cdr (car q)) (car q))
         (let ([x (caar q)])
           (set-cdr! q #f)
           x)]
        [(zero? n)
         (let ([x (caar q)])
           (let loop ([p (car q)])
             (if (eq? (cdr p) (car q))
                 (begin
                   (set-cdr! p (cdr (car q)))
                   (set-car! q p))
                 (loop (cdr p))))
           x)]
        [else (let loop ([n n] [q (car q)] [p (cdr (car q))])
                (if (zero? n)
                    (let ([x (car q)])
                      (set-cdr! q p)
                      x)
                    (loop (- n 1) (cdr q) (cdr p))))]))


(define (traverse-add a q)
  (cond [(null? q)
         (letrec ([x (cons a x)])
           (cons x #t))]
        [(q-traverse-empty? q)
         (set-cdr! q #t)
         (set-car! (car q) a)
         q]
        [else (set-cdr! (car q) (cons a (cdr (car q))))  q]))


(define traverse-empty?
  (lambda (q)
    (not (cdr q))))
```

Figure 7.8 Improved Node-list Operations for *Traverse*

```
(define (init-traverse) #f)

(define (run-traverse)
  (let ([*root* (create-structure 100)])
    (let loop ([i 50])
      (if (not (zero? i))
          (begin
            (traverse *root*)
            (traverse *root*)
            (traverse *root*)
            (traverse *root*)
            (traverse *root*)
            (loop (- i 1)))))))
```

Figure 7.9 Improved Driver Code for *Traverse*

less time than the original version. Figure 7.10 presents the types of some of the top-level definitions in the final program.

traverse-remove	:	(**rec** *([Y1 (cons X1 Y1)])* *(num (cons Y1 (+ false X2)) –> X1))*
traverse-select	:	(**rec** *([Y1 (cons X1 Y1)])* *(num (cons Y1 X2) –> X1))*
traverse-add	:	(**rec** *([Y1 (cons X1 Y1)])* *(X1 (+ nil (cons (cons X1 Y1) (+ true X2)))* *–> (cons Y1 (+ true X2))))*
traverse-empty?	:	*((cons X1 X2) –> (+ false true))*
create-structure	:	(**rec** *([Y1 (node (list Y1) (list Y1) num nil nil nil nil nil nil nil)])* *(num –> Y1))*
find-root	:	(**rec** *([Y1 (node (cons Y1 X1) X2 X3 X4 X5 X6 X7 X8 X9 X10)])* *(Y1 num –> Y1))*
traverse	:	(**rec** *([Y1 (node X1 (list Y1) X2 (+ false true X3) ...)])* *(Y1 X9 –> void))*
travers	:	(**rec** *([Y1 (node X1 (list Y1) X2 (+ false true X3) ...)])* *(Y1 –> num))*

Figure 7.10 Types for Improved *Traverse*

7.5.5 Improved Execution Times

Figure 7.11 presents the improved execution times for the four modified Gabriel bench marks. The light grey region indicates execution time that the original bench marks spent performing run-time checks, but which is eliminated in the modified programs. These results show that our prototype is an effective tool for safely eliminating run-time checking overhead from Scheme programs.

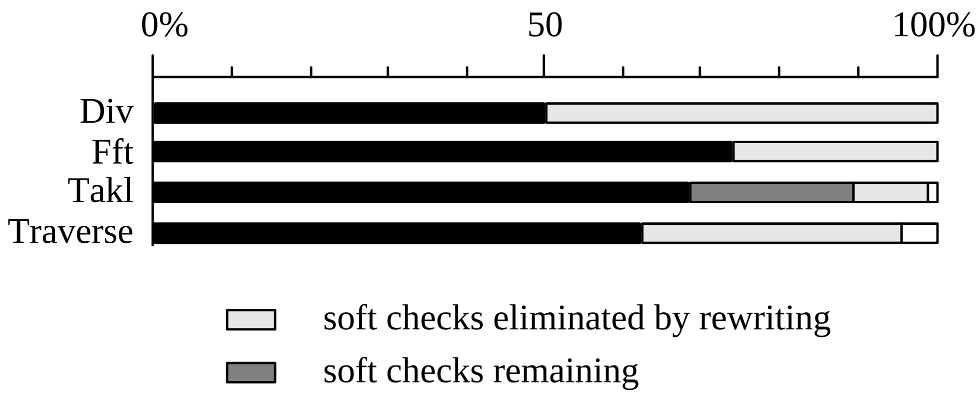

Figure 7.11 Improved Execution Times

Chapter 8

Problems and Future Work

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 soft type system that could be improved through either better soft typing technology or by changing the underlying programming language. There are also programming environment issues that our prototype does not address. Finally, the techniques used to implement soft typing may also be useful in the context of static type systems.

8.1 Typing Precision

As the problem of assigning precise static types to programs is undecidable, any type system must yield conservatively approximate types.[1] We believe that the intuitive models that 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.

8.1.1 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 apply a function to one or more 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

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

$$
\begin{array}{ll}
 & ((X1 \rightarrow X0)\ (list\ X1) \rightarrow X0) \\
\textbf{and} & ((X1\ X2 \rightarrow X0)\ (list\ X1)\ (list\ X2) \rightarrow X0) \\
\textbf{and} & ((X1\ X2\ X3 \rightarrow X0)\ (list\ X1)\ (list\ X2)\ (list\ X3) \rightarrow X0) \\
\textbf{and} & \ldots
\end{array}
$$

Dzeng and Haynes [15] adapt Rémy's encoding technique to give a precise type to variable-arity functions like Scheme's map. It seems straightforward to extend our soft type system to include their encoding. But as their system 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 a variable number of arguments, of which the last argument is special:

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

Given a function f, zero or more individual arguments $a_1 \ldots a_n$, and an argument list l, apply applies f to $a_1 \ldots a_n$ and the elements of l. 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 apply the internal type

$$((arg^{\varphi_1}\ ((arglist\ \alpha_1){\rightarrow}{*}^{\varphi_2}\alpha_2)\ (arg^{\varphi_3}\ (list\ \alpha_1)\ noarg)) \rightarrow{*}^{+}\ \alpha_2) \cup \phi,$$

where $(arglist\ \alpha_1)$ and $(list\ \alpha_1)$ 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. This run-time check can be eliminated by transforming the application $(\mathsf{apply}\ f\ a_1 \ldots a_n\ l)$ to $(\mathsf{apply}\ f\ (\mathsf{cons}\ a_1\ \ldots\ (\mathsf{cons}\ a_n\ l)))$. But apply's type also forces all arguments to be passed to f to have the same type. Hence the expression $(\mathsf{apply}\ f\ (\mathsf{cons}\ a_1\ \ldots\ (\mathsf{cons}\ a_n\ '())))$ is typed less precisely than the equivalent expression $(f\ a_1\ \ldots\ a_n)$.

Both of these problems 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 us from using it for argument lists because type constructors for mutable values must be treated differently from type constructors for immutable values in algorithms like type swapping (Section 3.7.2). If pairs were immutable, we could assign a reasonable type to apply by using the type constructors *cons* and *nil* rather than *arg* and *noarg* for argument lists. The internal type for apply would be

$$((cons^{\varphi_1}\ (\alpha_1 \text{−>*}^{\varphi_2} \alpha_2)\ \alpha_1)\ \text{−>*}^{+}\ \alpha_2) \cup \emptyset, \tag{8.1}$$

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 similar problems to apply. It takes a variable number of arguments, the last of which is special:

$$(\mathsf{append}\ l_1 \ldots l_n\ v).$$

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 often v is just another list). Soft Scheme assigns append the internal type

$$((arglist\ (list\ \alpha))\ \text{−>*}^{+}\ (list\ \alpha)) \cup \emptyset,$$

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 problem.

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.

8.1.2 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 3.3). Applications of **lambda**-expressions require the types of arguments to match the types the function expects (see rules **ap** and **Cap** in Figure 3.3). 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 following program:

```
((lambda (f y)
   (f y)
   (add1 y))
 (lambda (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 (**lambda** (x) (**if** x x #f)) is passed to f forces f to have type *((+ false T) –> (+ 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 *(+ 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 (lambda (x) (if x x #f))])
  ((lambda (y)
     (f y)
     (add1 y))
   1))
```

f is assigned the polymorphic internal type

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

Type swapping (see Section 3.7.2) replaces this type with the internal type

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

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 a more sophisticated approach to type inference than unification-based algorithms yield. To this end, we have investigated several adaptations of structural subtyping [32, 45]. 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 [53], particularly in the presence of recursive types.

Other possible solutions to the reverse flow problem are type inference methods based on constraint solving systems like that of Aiken *et al.* [3, 4] and Heintze [24]. Chapter 6 discusses these systems in more detail.

8.1.3 Assignment

Because assignment interferes with polymorphism, and therefore with subtyping, assignment can be a major source of imprecise types (see Section 5.1). 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. We see no solution to this problem other than carefully engineering programs to use assignable global 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.

8.2 Type Size

While our presentation types are concise and easy to understand, especially in comparison with types like those that Aiken's system [3, 4] infers, the types our system infers for procedures can still be quite large. Figure 5.5 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 tree = Leaf of int
              | Node of tree * tree
```

introduces constructors named `Leaf` and `Node`. Both constructors yield values of type *tree*. But adapting this solution to soft typing invokes the problems associated with type annotations 5.9. The constructor `Leaf` can only be applied to values of type *int*, and the constructor `Node` can only be applied to pairs of values of type *tree*. Hence constructor applications can require expensive or uncomputable run-time checks.

8.3 Explaining Run-time Checks

As Tofte [60: page 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 *not* will 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 (see Appendix C). As a last resort, the prototype can display internal types.

8.4 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 method for unboxing values in ML [36] 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, like 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 [8, 33, 39, 40] and some object-oriented systems [12, 34] use iterative flow analysis techniques to determine type information for optimization. The idea of using type information derived from type inference for optimization seems to have received little attention in the literature.

8.5 Programming Environment Issues

A good 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 proposed for separately compiling ML [54]. 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
  (lambda (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 (*i*) needs a run-time check,

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

(*iii*) 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.

8.6 Applications to Static Type Checking

Recall that in Chapter 1 we described static type systems as partitioning the data domain into disjoint *datatypes*, with values within datatypes constructed from overlapping *variants*. We showed that 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 like Standard ML. This type information could be used to eliminate variant checks and suppress unnecessary warning messages corresponding to the eliminated variant checks.

Appendix A

Proofs

A.1 Subject Reduction

Subject Reduction (Lemma 3.2) is the key lemma to proving type soundness for the static type system of Figure 3.3.

Some obvious facts about deductions that we use with no more ado are:

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

(ii) 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 (a) there is exactly one inference rule for each expression form, and (b) each inference rule requires a proof for each subexpression of the expression in its conclusion.

The following lemma states that extra variables in the type environment A of a judgment $A \vdash e : \tau$ that are not free in the expression e may be ignored.

Lemma A.1 If $A(x) = A'(x)$ for all $x \in FV(e)$, then $A \vdash e : \tau$ iff $A' \vdash e : \tau$.

A key lemma that we use in the proof of Subject Reduction is a Replacement Lemma, adapted from Hindley and Seldin [28: page 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 A.2 *(Replacement).* If:

(i) $\mathcal{D}$ is a deduction concluding $A \vdash C[e_1] : \tau$,

(ii) $\mathcal{D}_1$ is a subdeduction of $\mathcal{D}$ concluding $A' \vdash e_1 : \tau'$,

(iii) $\mathcal{D}_1$ occurs in $\mathcal{D}$ in the position corresponding to the hole in C, and

(iv) $A' \vdash e_2 : \tau'$,

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

Proof. We think of deductions as trees with the conclusion at the root. Let $\mathcal{D}_2$ be the deduction concluding $A' \vdash e_2 : \tau'$. Cut the subtree $\mathcal{D}_1$ out of $\mathcal{D}$ and replace it with $\mathcal{D}_2$. Replace all relevant occurrences of e_1 in the resulting tree with e_2. Then the resulting tree is a valid deduction concluding $A \vdash C[e_2] : \tau$, as may be shown by induction on the height of the tree [28: page 181]. $\square$

Lemma A.3 *(Subject Reduction).* If $\vdash e_a : \sigma$ and $e_a \longmapsto e_b$ then $\vdash e_b : \sigma$.

Proof. The proof proceeds by case analysis according to the reductions of Figure 3.1. 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, τ such that $A \vdash (\mathbf{ap}\ p\ v) : \tau$, we have $A \vdash p : (\tau' \to^f \tau) \cup \emptyset$ and $A \vdash v : \tau'$ by $\mathbf{ap}_\vdash$. By the typability condition (Definition 3.6), $A \vdash \delta(c, v) : \tau$, and hence $\vdash E[\delta(c, v)] : \sigma$ by *Replacement*.

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

Case $E[(\mathbf{if}\ v\ e_1\ e_2)] \longmapsto E[e_1]$ where $v \neq \#\mathsf{f}$. Since there exist A, τ 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, 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 $E[(\mathbf{ap}\ (\mathbf{lambda}\ (x)\ e)\ v)] \longmapsto E[e[x/v]]$. Since there exist A, τ such that $A \vdash (\mathbf{ap}\ (\mathbf{lambda}\ (x)\ e)\ v) : \tau$, we have $A \vdash v : \tau'$ and $A \vdash (\mathbf{lambda}\ (x)\ e) : (\tau' \to^f \tau) \cup \emptyset$ by $\mathbf{ap}_\vdash$. From the latter, $A[x \mapsto \tau'] \vdash e : \tau$ (and $f = \mathbf{+}$) by $\mathbf{lam}_\vdash$. Hence $A \vdash e[x/v] : \tau$ by Lemma A.4, and $\vdash E[e[x/v]] : \sigma$ by *Replacement*.

Case $E[(\mathbf{CHECK\text{-}ap}\ (\mathbf{lambda}\ (x)\ e)\ v)] \longmapsto E[e[x/v]]$. Similar to the previous case.

Case $E[(\mathbf{let}\ ([x\ v])\ e)] \longmapsto E[e[x/v]]$. 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 $A \vdash e[x/v] : \tau$ by Lemma A.4. Hence $\vdash E[e[x/v]] : \sigma$ by *Replacement*. $\square$

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

Lemma A.4 *(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 $A \vdash e[x/v] : \tau'$.

Proof. We proceed by induction on the length of the proof of $A[x \mapsto \forall\vec{\alpha}\vec{\varphi}.\tau] \vdash e : \tau'$, and case analysis on the last step. The proof is an adaptation of our similar lemma for an ordinary Hindley-Milner type system [64: lemma 4.4]. $\square$

A.2 Universal Applicability

Universal Applicability (part (i) of Lemma 3.2) states that we can find a soft typing in the soft type system of Figure 3.4 for any program. *Universal Applicability* relies on the fact that any two types can be unified.

Lemma A.5 For any τ_1, τ_2 there exists S such that $S\tau_1 = S\tau_2$.

The proof of this lemma depends on the fact that soft types do not use $\mathbf{-}$ or $\not{\phi}$. More generally, we can show that for any set of pairs of types $\{\langle \tau_i, \tau_i' \rangle\}$, there is a substitution that unifies τ_i and τ_i' of every pair.

To prove *Universal Applicability*, we also require that soft typings be stable under substitution.

Lemma A.6 If $A \models e \Rightarrow e' : \tau$ and S is a substitution then there exists e'' such that $SA \models e \Rightarrow e'' : S\tau$.

Proof. The proof proceeds by induction on the length of the proof of $A \models e \Rightarrow e' : \tau$ and case analysis on the last step. The proof is an adaptation of Tofte's proof of a similar lemma [60: lemma 4.2]. $\square$

Given these lemmas, we can show that a soft typing deduction can be obtained for any program.

Lemma A.7 *(Universal Applicability).* For all programs e, there exist e', τ such that $\models e \Rightarrow e' : \tau$.

Proof. We prove the following stronger statement.

For all e and all A such that $FV(e) \subseteq \mathrm{Dom}(A)$, there exist e', τ such that $A \models e \Rightarrow e' : \tau$.

The proof proceeds by induction on the height of a deduction and case analysis on the structure of expression e.

Case c. We can use either $\mathbf{OKconst}_{\models}$ or $\mathbf{const}_{\models}$ to get either $A \models c \Rightarrow c : \tau$ or $A \models c \Rightarrow \mathsf{CHECK}\text{-}c : \tau$ where $\tau \prec_S \mathit{SoftTypeOf}(c)$.

Case x. By $\mathbf{id}_{\models}$ $A \models x \Rightarrow x : \tau$ where $\tau \prec_S A(x)$.

Case $(\mathbf{ap}\ e_1\ e_2)$. Using the induction hypothesis twice:

$$A \models e_1 \Rightarrow e_1' : \tau_1 \tag{A.1}$$
$$\text{and}\quad A \models e_2 \Rightarrow e_2' : \tau_2. \tag{A.2}$$

By Lemma A.5 we can find S_1 such that $S_1\tau_1 = (\tau_3 \to^f \tau_4) \cup \tau_5$ for any $\tau_3, \tau_4, \tau_5, f$. Applying Lemma A.6 to (A.1) and (A.2):

$$S_1 A \models e_1 \Rightarrow e_1'' : (\tau_3 \to^f \tau_4) \cup \tau_5 \tag{A.3}$$
$$\text{and}\quad S_1 A \models e_2 \Rightarrow e_2'' : S_1\tau_2. \tag{A.4}$$

Again by Lemma A.5 we can find S_2 such that $S_2 S_1 \tau_2 = S_2\tau_3$. Applying Lemma A.6 to (A.3) and (A.4):

$$S_2 S_1 A \models e_1 \Rightarrow e_1''' : (S_2\tau_3 \to^{S_2 f} S_2\tau_4) \cup S_2\tau_5$$
$$\text{and}\quad S_2 S_1 A \models e_2 \Rightarrow e_2''' : S_2\tau_3.$$

By either $\mathbf{OKap}_{\models}$ or $\mathbf{CHECK\text{-}ap}_{\models}$, according to whether $S_2\tau_5$ is an absent type, we get either:

$$S_2 S_1 A \models (\mathbf{ap}\ e_1\ e_2) \Rightarrow (\mathbf{ap}\ e_1'''\ e_2''') : S_2\tau_4$$
$$\text{or}\quad S_2 S_1 A \models (\mathbf{ap}\ e_1\ e_2) \Rightarrow (\mathbf{CHECK\text{-}ap}\ e_1'''\ e_2''') : S_2\tau_4.$$

Case $(\mathbf{lambda}\ (x)\ e)$. By the induction hypothesis:

$$A[x \mapsto \tau_1] \models e \Rightarrow e' : \tau_2.$$

Then $A \models (\mathbf{lambda}\ (x)\ e) \Rightarrow (\mathbf{lambda}\ (x)\ e') : (\tau_1 \to^{+} \tau_2) \cup \tau_3$ by $\mathbf{lam}_{\models}$.

Case (**if** e_1 e_2 e_3). Using the induction hypothesis three times:

$$\begin{aligned} & A \models e_1 \Rightarrow e_1' : \tau_1 & (A.5) \\ \text{and} \quad & A \models e_2 \Rightarrow e_2' : \tau_2 & (A.6) \\ \text{and} \quad & A \models e_3 \Rightarrow e_3' : \tau_3. & (A.7) \end{aligned}$$

By Lemma A.5 we can find S_1 such that $S_1\tau_2 = S_1\tau_3$. Applying Lemma A.6 to (A.5), (A.6), and (A.7):

$$\begin{aligned} & S_1 A \models e_1 \Rightarrow e_1'' : S_1\tau_1 \\ \text{and} \quad & S_1 A \models e_2 \Rightarrow e_2'' : S_1\tau_2 \\ \text{and} \quad & S_1 A \models e_3 \Rightarrow e_3'' : S_1\tau_2. \end{aligned}$$

Then $S_1 A \models (\textbf{if}\ e_1\ e_2\ e_3) \Rightarrow (\textbf{if}\ e_1''\ e_2''\ e_3'') : S_1\tau_2$ by $\textbf{if}_{\models}$.

Case (**let** ([x e_1]) e_2). Using the induction hypothesis twice:

$$\begin{aligned} & A \models e_1 \Rightarrow e_1' : \tau_1 \\ \text{and} \quad & A[x \mapsto \Sigma] \models e_2 \Rightarrow e_2' : \tau_2 \end{aligned}$$

for any Σ. Let $\Sigma = \mathit{SoftClose}(\tau_1, A)$. Then:

$$A \models (\textbf{let}\ ([x\ e_1])\ e_2) \Rightarrow (\textbf{let}\ ([x\ e_1'])\ e_2') : \tau_2$$

by $\textbf{let}_{\models}$. □

A.3 Static Typability

Static Typability (Lemma 3.5) indicates that the program with run-time checks obtained by a soft typing has a type in the static type system. (Recall that $\tilde{S}$ is the substitution with domain *AbsVar* that takes all absent type variables to $\emptyset$ and all absent flag variables to **–**.)

Lemma A.8 *(Static Typability).* If $A \models e \Rightarrow e' : \tau$ then $\tilde{S}A \vdash e' : \tilde{S}\tau$.

Proof. The proof proceeds by induction over the height of the deduction $A \models e \Rightarrow e' : \tau$ and case analysis on the last rule.

Case $A \models c \Rightarrow c : \tau$. By $\mathbf{OKconst}_{\models}$ we have $\tau \prec_S \mathit{SoftTypeOf}(c)$ and $\mathit{empty}\{S\tilde{\nu} \mid \tilde{\nu} \in \mathrm{Dom}(S)\}$. Let $\mathit{SoftTypeOf}(c) = \forall \vec{\nu}_1.\, \tau_1$ and $\mathit{TypeOf}(c) = \forall \vec{\nu}_2.\, \tau_2$. By the definitions of *SoftTypeOf* and *TypeOf*, $\{\vec{\nu}_2\} = \{\vec{\nu}_1\} - \mathit{AbsVar}$ and $\tilde{S}\tau_1 = \tau_2$. That is, we have the following situation:

$$\begin{tikzcd} \tau_1 \arrow[r, "S"] \arrow[d, "\tilde{S}"'] & \tau \arrow[d, "\tilde{S}"] \\ \tau_2 \arrow[r, dashed, "S'"] & \tilde{S}\tau \end{tikzcd}$$

To obtain $\tilde{S}\tau \prec_{S'} \mathit{TypeOf}(c)$, we must find S' with domain $\{\vec{\nu}_2\}$ such that the diagram commutes. Let $S' = \{\nu/\tilde{S}S\nu \mid \nu \in \{\vec{\nu}_2\}\}$, and consider the following three cases:

(i) ν is a non-absent variable in τ_1. Then $\tilde{S}\nu = \nu$, hence $S'\tilde{S}\nu = S'\nu = \tilde{S}S\nu$.

(ii) ν is an absent flag variable in τ_1. Then $S'\tilde{S}\nu = S'\mathbf{-} = \mathbf{-}$. Also $\tilde{S}S\nu = \mathbf{-}$ since $\mathit{empty}(S\nu)$.

(iii) ν is an absent type variable in τ_1. Then $S'\tilde{S}\nu = S'\emptyset = \emptyset$. Also $\tilde{S}S\nu = \emptyset$ since $\mathit{empty}(S\nu)$.

Hence the diagram commutes and we have $\tilde{S}\tau \prec_{S'} \mathit{TypeOf}(c)$. Hence $\tilde{S}A \vdash c : \tilde{S}\tau$ by $\mathbf{const}_{\vdash}$.

Case $A \models c \Rightarrow \mathsf{CHECK}\text{-}c : \tau$. We have $\tau \prec_S \mathit{SoftTypeOf}(c)$ by $\mathbf{const}_{\models}$. Let $\mathit{SoftTypeOf}(c) = \forall \vec{\nu}_1.\, \tau_1$ and $\mathit{TypeOf}(c) = \forall \vec{\nu}_2.\, \tau_2$. By the definitions of $\mathit{SoftTypeOf}(c)$ and $\mathit{TypeOf}(\mathsf{CHECK}\text{-}c)$, there exists a bijective substitution B with domain $\{\vec{\nu}_1\} \cap \mathit{AbsVar}$ and range $\{\vec{\nu}_2\} - \{\vec{\nu}_1\}$ such that $B\tau_1 = \tau_2$. That is, we have the following situation:

$$\begin{tikzcd} \tau_1 \arrow[r, "S"] \arrow[d, "B"'] & \tau \arrow[d, "\tilde{S}"] \\ \tau_2 \arrow[r, dashed, "S'"] & \tilde{S}\tau \end{tikzcd}$$

To obtain $\tilde{S}\tau \prec_{S'} \mathit{TypeOf}(c)$, we must find S' with domain $\{\vec{\nu}_2\}$ such that the diagram commutes. Let $S' = \{\nu/\tilde{S}SB^{-1}\nu \mid \nu \in \{\vec{\nu}_2\}\}$. Then $S'B\nu =$

$\tilde{S}SB^{-1}B\nu = \tilde{S}S\nu$. Hence the diagram commutes and $\tilde{S}\tau \prec_{S'} \mathit{TypeOf}(c)$. Hence $\tilde{S}A \vdash c : \tilde{S}\tau$ by $\mathbf{const}_{\vdash}$.

Case $A \models\!\!\!\!\!> x \Rightarrow x : \tau$. We have $\tau \prec_S A(x)$ by $\mathbf{id}_{\models\!\!>}$. Let $A(x) = \forall\vec{\nu}_1.\,\tau_1$ where $\vec{\nu}_1 = \mathrm{Dom}(S)$. Then $S\tau_1 = \tau$, hence $\tilde{S}S\tau_1 = \tilde{S}\tau$. As neither $\mathbf{lam}_{\models\!\!>}$ nor $\mathbf{let}_{\models\!\!>}$ introduce type schemes for identifiers with bound absent variables, $\{\vec{\nu}_1\} \not\cap \mathit{AbsVar}$. Hence $\tilde{S}S\tilde{S}\tau_1 = \tilde{S}\tau$. Let $S' = \tilde{S} \circ S$. Then $S'\tilde{S}\tau_1 = \tilde{S}\tau$, hence $\tilde{S}\tau \prec_{S'} \tilde{S}A(x)$. Hence $\tilde{S}A \vdash x : \tilde{S}\tau$ by $\mathbf{id}_{\vdash}$.

Case $A \models\!\!\!\!\!> (\mathbf{ap}\ e_1\ e_2) \Rightarrow (\mathbf{ap}\ e_1'\ e_2') : \tau_1$. Then $A \models\!\!\!\!\!> e_1 \Rightarrow e_1' : (\tau_2 \rightarrow^f \tau_1) \cup \tilde{\tau}_3$ and $A \models\!\!\!\!\!> e_2 \Rightarrow e_2' : \tau_2$ and $\mathit{empty}(\tilde{\tau}_3)$ by $\mathbf{OKap}_{\models\!\!>}$. Using induction on each subdeduction:

$$\tilde{S}A \vdash e_1' : (\tilde{S}\tau_2 \rightarrow^{\tilde{S}f} \tilde{S}\tau_1) \cup \tilde{S}\tilde{\tau}_3 \quad \text{and} \quad \tilde{S}A \vdash e_2' : \tilde{S}\tau_2.$$

But $\tilde{S}\tilde{\tau}_3 = \emptyset$ because $\mathit{empty}(\tilde{\tau}_3)$, so $\tilde{S}A \vdash (\mathbf{ap}\ e_1'\ e_2') : \tilde{S}\tau_1$ by $\mathbf{ap}_{\vdash}$.

Case $A \models\!\!\!\!\!> (\mathbf{let}\ ([x\ e_1])\ e_2) \Rightarrow (\mathbf{let}\ ([x\ e_1'])\ e_2') : \tau_2$. Then $A \models\!\!\!\!\!> e_1 \Rightarrow e_1' : \tau_1$ and $A[x \mapsto \mathit{SoftClose}(\tau_1, A)] \models\!\!\!\!\!> e_2 \Rightarrow e_2' : \tau_2$ by $\mathbf{let}_{\models\!\!>}$. By induction with each:

$$\tilde{S}A \vdash e_1' : \tilde{S}\tau_1 \quad \text{and} \quad (\tilde{S}A)[x \mapsto \tilde{S}\mathit{SoftClose}(\tau_1, A)]) \vdash e_2' : \tilde{S}\tau_2.$$

Let $\mathit{SoftClose}(\tau_1, A) = \forall\vec{\nu}.\,\tau_1$ where $\{\vec{\nu}\} = FV(\tau_1) - (FV(A) \cup \mathit{AbsVar})$. Then $\tilde{S}\forall\vec{\nu}.\,\tau_1 = \forall\vec{\nu}.\,\tilde{S}\tau_1$. Since:

$$\begin{aligned}
& FV(\tilde{S}\tau_1) - FV(\tilde{S}A) \\
= \ & (FV(\tau_1) - \mathit{AbsVar}) - (FV(A) - \mathit{AbsVar}) \\
= \ & FV(\tau_1) - (FV(A) \cup \mathit{AbsVar}) \\
= \ & \{\vec{\nu}\},
\end{aligned}$$

$\tilde{S}\mathit{SoftClose}(\tau_1, A) = \mathit{Close}(\tilde{S}\tau_1, \tilde{S}A)$. Then $(\tilde{S}A)[x \mapsto \mathit{Close}(\tilde{S}\tau_1, \tilde{S}A)] \vdash e_2' : \tilde{S}\tau_2$, hence $\tilde{S}A \vdash (\mathbf{let}\ ([x\ e_1'])\ e_2') : \tilde{S}\tau_2$ by $\mathbf{let}_{\vdash}$.

The remaining cases are straightforward. □

Appendix B

Semantics of Types

We use operational semantics throughout this thesis because we believe that language semantics and proofs of type soundness are most clearly expressed in an operational framework [64]. 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 *Pure 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. We refrain from developing a complete denotational semantics for *Pure Scheme* and showing that it corresponds to our operational semantics [49].

B.1 Internal Types

We use the ideal model developed by MacQueen, Plotkin, and Sethi [41] to assign meaning to the internal types of Chapter 3.

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

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

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

The meaning of a type τ is an *ideal* over $\mathbb{D}$. An ideal is a set of values that is closed

(i) downwards under approximations, and

(ii) upwards to least upper bounds of consistent sets of values.

Let $\mathcal{I}(\mathbb{D})$ be the set of ideals of $\mathbb{D}$. 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 $\mathbb{D}$, and the free flag variables to the set $\{\mathbf{+}, \mathbf{-}\}$:

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

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

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

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 $\mathcal{T}$: $\mathit{Type} \rightarrow \mathit{TypeEnv} \rightarrow \mathcal{I}(\mathbb{D})$ defined in Figure B.1 maps a type to an ideal of $\mathbb{D}$. We define the product ($\boxed{\times}$) and exponentiation ($\boxed{\rightarrow}$) functions on ideals as follows:

$$\begin{aligned} I \boxed{\times} J \;&=\; I \otimes J \\ I \boxed{\rightarrow} J \;&=\; \{f \in \mathbb{D} \rightarrow_{sc} \mathbb{D} \;\mid\; f(I) \subseteq J\} \end{aligned}$$

and we identify their results with the corresponding subsets of $\mathbb{D}$.[1] Where $f : \mathcal{I}(\mathbb{D}) \rightarrow \mathcal{I}(\mathbb{D})$ is a function of one argument, let

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

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

[1] Technically, $I \boxed{\times} J$ and $I \boxed{\rightarrow} J$ are only *isomorphic* to corresponding subsets of $\mathbb{D}$, see MacQueen *et al.* [41: page 104].

B.2 Presentation Types

We use the same model to assign meaning to the presentation types of Chapter 4.

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

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

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 $\mathcal{P}$: $\mathit{Type} \rightarrow \mathit{TypeEnv} \rightarrow \mathcal{I}(\mathbb{D})$ defined in Figure B.2 maps a presentation type to an ideal of $\mathbb{D}$. The extra occurrences of $\mathbb{E}_\perp$ that do not arise in the semantics for internal types ensure that check is a member of singular union types like num. In the semantics for internal types, the corresponding type $\mathit{num}^{+} \cup \phi$ includes check because ϕ 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:

$$\begin{aligned}
&\mathcal{R}[\![(\mathbf{rec}\ ()\ T)]\!] \;=\; T\\
&\mathcal{R}[\![(\mathbf{rec}\ ([X_1\ T_1])\ T_0)]\!] = T_0[X_1/\mu X_1.\, T_1]\\
&\mathcal{R}[\![(\mathbf{rec}\ ([X_1\ T_1][X_2\ T_2]\ldots)\ T_0)]\!]\\
&\quad = \mathcal{R}[\![(\mathbf{rec}\ ([X_2\ T_2]\ldots)\ T_0)]\!][X_1/\mu X_1.\, \mathcal{R}[\![(\mathbf{rec}\ ([X_2\ T_2]\ldots)\ T_1)]\!]]
\end{aligned}$$

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

B.3 Translating to Presentation Types

Chapter 4 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.1 If T is the presentation form of closed internal type τ, then $\mathcal{T}[\![\tau]\!]\emptyset \subseteq \mathcal{P}[\![T]\!]\emptyset$.

$$
\begin{array}{rcl}
\mathcal{T}[\![\emptyset]\!]\rho & = & \mathbb{E}_\perp \\
\mathcal{T}[\![num^f \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathbb{N}_\perp \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![true^f \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathbb{T}_\perp \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![false^f \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathbb{F}_\perp \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![nil^f \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathbb{I}_\perp \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![(cons^f\; \sigma_1\; \sigma_2) \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathcal{T}[\![\sigma_1]\!]\rho \boxed{\times} \mathcal{T}[\![\sigma_2]\!]\rho \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![(\sigma_1 \to^f \sigma_2) \;\cup\; \tau]\!]\rho & = & (\text{if } \rho(f) = \mathbf{+} \text{ then } \mathcal{T}[\![\sigma_1]\!]\rho \boxed{\to} \mathcal{T}[\![\sigma_2]\!]\rho \text{ else } \emptyset) \cup \mathcal{T}[\![\tau]\!]\rho \\
\mathcal{T}[\![\alpha]\!]\rho & = & \rho(\alpha) \\
\mathcal{T}[\![\mu\alpha.\, \tau]\!]\rho & = & \mu(\lambda I \in \mathcal{I}(\mathbb{D}).\, \mathcal{T}[\![\tau]\!](\rho[\alpha \mapsto I])) \\
\mathcal{T}[\![\forall\alpha^{\{\kappa_1,\ldots,\kappa_n\}}.\, \tau]\!]\rho & = & \forall_{\mathcal{U}}(\lambda I \in \mathcal{I}(\mathbb{D}).\, \mathcal{T}[\![\tau]\!](\rho[\alpha \mapsto I])) \\
 & & \text{where } \mathcal{U} = \mathbb{D}_{\neg\{\kappa_1,\ldots,\kappa_n\}} \\
\mathcal{T}[\![\forall\varphi.\, \tau]\!]\rho & = & \mathcal{T}[\![\tau]\!](\rho[\varphi \mapsto \mathbf{+}]) \;\cap\; \mathcal{T}[\![\tau]\!](\rho[\varphi \mapsto \mathbf{-}])
\end{array}
$$

Figure B.1 Ideal Semantics for Internal Types

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

Figure B.2 Ideal Semantics for Presentation Types

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

(i) Replace all useless flag variables in τ with **-** and all useless type variables in τ with $\emptyset$. Let the result of this step be $\tau' = \mathit{useless}(\tau)$.

(ii) Replace all other flag variables with **+**. Let the result of this step be $\tau'' = \mathit{unflag}(\tau')$.

(iii) 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 (resp. type) variables that occur only positively in τ with **-** (resp. $\emptyset$). We can show by induction on the depth of nesting within the first argument of function type 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 **+**. That is, $\mathit{unflag}(\forall\varphi.\,\tau) = \tau[\varphi/\mathbf{+}]$. We show that $\mathcal{T}[\![\forall\varphi.\,\tau]\!]\emptyset \subseteq \mathcal{T}[\![\tau[\varphi/\mathbf{+}]]\!]\emptyset$ as follows:

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

This completes the proof. $\square$

Appendix C

Soft Scheme Reference Manual

This chapter contains the reference manual for Soft Scheme, a soft type checker for R4RS Scheme. Soft Scheme is available by anonymous FTP from cs.rice.edu in public/wright/match.tar.Z. It works best with Chez Scheme (version 4.1), but should be easy to port to other Scheme systems.

C.1 Preliminaries

Soft Scheme defines several commands as well as run-time checks for every R4RS primitive. Commands begin with the prefix "st:". Run-time checks begin with the prefix "CHECK-". For example, the run-time check for map is CHECK-map. There are also a few macros with a similar naming convention. Hence programs to be soft typed should avoid using identifiers beginning with "st:" or "CHECK-".

Soft Scheme supports all of R4RS Scheme, with only a few limitations as listed in Section C.6. The most frequently encountered limitation is that applications of the procedure make-vector cannot omit the second argument which supplies the initial value for elements. Soft Scheme also supports several extensions to R4RS Scheme, as listed in Section C.4, including pattern matching and data definition.

Soft Scheme generates various informative messages. Less important messages are introduced by "Note"; more important messages are preceded by "Warning". Warnings include messages about unbound identifiers because the type checker cannot guarantee the safety of programs with unbound identifiers.

Soft Scheme is a batch system. Given a Scheme program, Soft Scheme produces type information for that program and an annotated program with run-time checks inserted. The command st: type checks a file, writes an annotated version of the program with checks inserted to an output file, prints a summary of the inserted checks, and loads and executes the program provided that it has no unbound references. A program consists of the usual Scheme definitions and expressions, and may be spread across several files. When a program (called the *current program*) is type checked, the type checker remembers the types of its top-level definitions. These types may be inspected with the command **st:type**.

In developing soft typed programs, it is often desirable to type check a fragment of a program. Soft Scheme assumes type $\forall\alpha.\,\alpha$ for any unbound identifiers.

C.2 Commands

Following are the commands available in version 0.10 of Soft Scheme. The notation [*item*] indicates that *item* is optional. The notation *item* ... indicates zero or more occurrences of *item*.

- (**st:** *input* [*output*])

input	is a file name
output	is a file name or #f

  Type checks the program. If *output* is present and is not #f, an annotated version of the program is written to that file. If *output* is missing, the annotated version is written to a file name constructed from the input file name with suffix ".soft". The command **st:summary** is invoked to report the inserted checks. Finally, if there were no unbound references, the output is loaded and executed by **st:run**.

- (**st:** '(*input* ...) *output*)

  Like above, but the input program may be spread over multiple files. An output file name is required.

- (**st:** *expression* [*output*])

expression	is any Scheme expression or definition

  Like above, but the input may be a top-level Scheme expression, quoted as an s-expression.

- (**st:check** *input* [*output*])
  (**st:check** '(*input* ...) *output*)
  (**st:check** *expression* [*output*])

  Like synst:, but **st:run** is not called to execute the program.

- (**st:run** *output*)

output is a file name

Loads and executes a previously generated soft typed program.

- (**st:summary**)

  Prints a summary of run-time checks for the current program. For each global *definition* that requires at least one run-time check, the summary includes a line of the form:

  definition N [(M match [I inexhaustive])]
  [(P prim)]
  [(F field)]
  [(L lambda)]
  [(A ap)]
  [(T TYPE)]
  [(E ERROR)]

  There are six kinds of run-time checks: match checks, primitive checks, field checks, lambda checks, application checks, and type checks. N is the sum of all six kinds of checks. The numbers M, P, F, L, A, and T indicate the distribution of checks. The most common kind of check is a primitive check. Other kinds of checks frequently indicate programming errors. Section C.3 explains the various kinds of run-time checks in more detail.

  Some checks are classified as *errors*. If reached, such a check cannot succeed. E indicates the number of such checks, if any. Following the summaries for individual definitions is a total line:

  TOTAL CHECKS N (of possible-checks is P %)
  [(ERROR E)] [(TYPE T)]

  This line reports the total number of run-time checks required for the entire program, the potential number of checks (how many a naive Scheme compiler inserts), the percentage P of sites that require run-time checks, the total number of errors, if any, and the total number of failed type annotations, if any.

- (**st:type** *definition* ...)

  definition is the name of a top-level definition

  Prints the types of top-level definitions of the current program. The definition names must be quoted symbols. This command can also be used to inspect the types of primitive operations, like (st:type 'force).

- (**st:type**)

Prints the type of every top-level definition in the current program.

- (**st:type** N ...)

  N is a CHECK or CLASH number

  Prints the types of each CHECK or CLASH. See st:cause.

- (**st:ltype** *definition* ...)

  definition is the name of a top-level definition

  Prints the types of all local definitions within each named top-level definition.

- (**st:ltype**)

  Prints the types of every local definition in the entire program.

- (**st:cause**)

  Prints cause information for CHECKs. This command is intended to help locate why a run-time CHECK is inserted. For each inserted CHECK, st:cause prints a list of cause items. There are two kinds of cause items:

  1. A cause item may be a set of identifiers, like (even? odd?), one of which contains the CHECK. This item indicates that even? and odd? are mutually recursive and are type checked (generalized) together. Independent of their uses, the definitions of these procedures force the check. Command (st:type N) where N is the CHECK identifier number will print the type of the checked primitive. This type is determined only from the definitions of even? and odd?, and will include some elements that the unchecked primitive does not accept.
  2. A cause item may be a CLASH number. A CLASH is a use of a polymorphic non-primitive procedure that forces a CHECK to be inserted. In this case, (st:type N) where N is the CHECK number may not include elements that the unchecked primitive does not accept. In this case, the type of the primitive indicates the type it would have were the CLASH not present. But (st:type N) where N is the CLASH number will display the type of the CLASH.

  If a check has several causes, each of these causes may need to be repaired to eliminate the check. To better understand st:cause, try it with the example referred to at end of this document.

- (**st:control** *flag* ...)

  flag is '+raw, '-raw, '+verbose, or '-verbose

  Sets various internal modes. Flag '+verbose causes definition names to be printed as they are type checked. The flag '+raw causes raw types to be displayed. There are other flags not intended for general use.

- (**st:help**)

  Prints a brief summary of these commands.

C.3 The Annotated Program

The annotated program includes run-time checks to guarantee safe program execution. This annotated program can be run at Chez Scheme's (optimize-level 3) with no possibility of a memory violation, provided there were no unbound references.[1]

Every run-time check has a unique identification number. This number is reported in the error message when a run-time check fails. By searching for this number in the annotated program, it is easy to determine exactly which run-time check failed.

The six kinds of run-time checks are:

(CHECK-match (# [INEXHAUST]) *exp clauses ...*)
(CHECK-*primitive* # [ERROR])
(CHECK-ap (# [ERROR]) *fun args ...*)
(CHECK-lambda (# [ERROR]) *names body ...*)
(CHECK-field (#) *name exp*)
(CHECK-: (#) *type exp*)

where *primitive* is the name of a primitive operation. In each case, # is the identification number of the check. A primitive check like CHECK-car is inserted if the arguments of the primitive may not be the right type, or if the wrong number of arguments is present. CHECK-match is inserted for a match check. Match checks are further classified as ordinary or *inexhaustive*. A match is inexhaustive if the type system cannot precisely represent its possible input. For example, the expression (**match** x [1 ...]) is inexhaustive because the type system must use the approximate type *number* for its input. An ordinary match check indicates that the type of the input value is larger than the type assigned by the system to the match expression,

[1] And provided programs do not rely on the order of evaluation of **letrec** expressions, see Section C.6.

even if this type is not precise. Hence (**match** #t [1 ...]) gets an ordinary match check. CHECK-ap is inserted when the function expression may not be a procedure. CHECK-lambda is generated for lambda expressions that may be called with the wrong number of arguments. CHECK-field is generated for field operations whose record argument may not possess the required field. Finally, CHECK-: is inserted for a failed type annotation. CHECK-: is guaranteed to terminate execution if reached. CHECK-: run-time checks are reported in the summary of run-time checks as TYPE checks. Any check that includes ERROR is guaranteed to terminate execution if reached.

C.4 Extensions to Scheme

C.4.1 Data Definition

Soft Scheme provides a slightly altered version of Chez Scheme's **define-structure** macro for defining new forms of data, and a similar **define-const-structure** form for defining immutable data.

The following expression defines a new kind of data named *struct*:

(**define-structure** (*struct* $arg_1 \ldots arg_n$))

A *struct* is a composite data structure with n *fields* named arg_1 through arg_n. The **define-structure** form declares the following procedures for constructing and manipulating data of type *struct*:

Procedure Name:	*Function*:
make-*struct*	constructor requiring n arguments
struct?	predicate
struct-arg$_1$, ..., *struct-arg*$_n$	named selectors
set-*struct-arg*$_1$!, ..., set-*struct-arg*$_n$!	named mutators

The field name _ (underscore) is special: no named selectors or mutators are defined for such a field. Such unnamed fields can only be accessed through pattern matching.

A second form of definition:

(**define-const-structure** (*struct* $arg_1 \ldots arg_n$))

is similar to **define-structure**, but allows immutable fields. If a field name arg_i is simply an identifier, no mutator is declared for that field. If a field name has the form (! x) where x is an identifier, then that field is mutable. Thus (**define-structure** (Foo a b)) abbreviates (**define-const-structure** (Foo (! a) (! b))).

Structures are implemented as vectors whose first component is the name of the structure as a symbol. Programs must not rely on or exploit this representation as our type checker assumes that structures are abstract objects.

C.4.2 Pattern Matching

Pattern matching allows complicated control decisions based on data structure to be expressed in a concise manner. Pattern matching is found in several modern languages, notably Standard ML, Haskell and Miranda.

The basic form of pattern matching expression is:

(**match** *exp* [*pat body*] ...)

where *exp* is an expression, *pat* is a pattern, and *body* is one or more expressions (like the body of a **lambda**-expression). The **match** form matches its first subexpression against a sequence of patterns, and branches to the *body* corresponding to the first pattern successfully matched.

Pattern Matching Expressions

The complete syntax of the pattern matching expressions accepted by Soft Scheme follows:

exp	::=	(**match** *exp clause* ...)
	|	(**match-lambda** *clause* ...)
	|	(**match-lambda*** *clause* ...)
	|	(**match-let** ((*pat exp*) ...) *body*)
	|	(**match-let*** ((*pat exp*) ...) *body*)
	|	(**match-letrec** ((*pat exp*) ...) *body*)
	|	(**match-let** var ((*pat exp*) ...) *body*)
clause	::=	(*pat body*) | (*pat* (`=>` *identifier*) *body*)

The **match-lambda** and **match-lambda*** forms are convenient combinations of **match** and **lambda**, and can be explained as follows:

(**match-lambda** [*pat body*] ...)	=	(**lambda** (x) (**match** x [*pat body*] ...))
(**match-lambda*** [*pat body*] ...)	=	(**lambda** x (**match** x [*pat body*] ...))

where x is a unique identifier. The **match-lambda** form is convenient when defining a single argument function that immediately destructures its argument. The

match-lambda* form constructs a function that accepts any number of arguments; the patterns of **match-lambda*** should be lists.

The **match-let**, **match-let***, **match-letrec**, and **match-define** expression forms generalize Scheme's **let**, **let***, **letrec**, and **define** expressions to allow patterns in the binding position rather than just identifiers. These forms are convenient for destructuring the result of a function that returns multiple values. As usual for **letrec** and **define**, pattern identifiers bound by **match-letrec** and **match-define** should not be used in computing the bound value.

The **match**, **match-lambda**, and **match-lambda*** forms allow the optional syntax (`=>` identifier) between the pattern and the body of a clause. When the pattern match for such a clause succeeds, the *identifier* is bound to a *failure procedure* of zero arguments within the *body*. If this procedure is invoked, it jumps back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. The *body* must not mutate the object being matched, otherwise unpredictable behavior may result.

Patterns

Following is the full syntax for patterns accepted by Soft Scheme.

		Pattern :	*Matches :*
pat	::=	*identifier*	anything, and binds *identifier*
	|	_	anything
	|	()	itself (the empty list)
	|	#t	itself
	|	#f	itself
	|	*string*	an *equal?* string
	|	*number*	an *equal?* number
	|	*character*	an *equal?* character
	|	'*s-expression*	an *equal?* s-expression
	|	'*symbol*	an *equal?* symbol (special case of s-expression)
	|	$(pat_1 \ldots pat_n)$	a proper list of n elements
	|	$(pat_1 \ldots pat_n\ .\ pat_{n+1})$	a list of n or more elements
	|	$\#(pat_1 \ldots pat_n)$	a vector of n elements
	|	#&*pat*	a box
	|	(**$** *struct* $pat_1 \ldots pat_n$)	a structure
	|	(**and** pat_1 pat_2)	if pat_1 and pat_2 match
	|	(**?** *predicate* [*pat*])	if *predicate* true and *pat* matches

Explanations of these patterns follow.

identifier **(excluding the reserved names ?, $, _, and and):** matches anything and binds an identifier of this name to the matching value in the *body*.

_ (underscore): matches anything, without binding any identifiers.

(), #t, #f, *string*, *number*, *character*, *'s-expression*: These constant patterns match themselves, *ie.,* the corresponding value must be *equal?* to the pattern.

$(pat_1 \ldots pat_n)$: matches a proper list of n elements that match pat_1 through pat_n.

$(pat_1 \ldots pat_n\ .\ pat_{n+1})$: matches a (possibly improper) list of at least n elements that ends in something matching pat_{n+1}.

$\#(pat_1 \ldots pat_n)$: matches a vector of length n, whose elements match pat_1 through pat_n.

$\#\&pat$: matches a box containing something matching *pat*.

(**$** *struct* $pat_1 \ldots pat_n$): matches a structure declared with **define-structure**. See Section C.4.1.

(**and** pat_1 pat_2): matches if both pat_1 and pat_2 match. One subpattern must be an identifier.

(**?** *predicate* [*pat*]): In this pattern, *predicate* must be an identifier denoting a primitive predicate or a predicate introduced by **define-structure**. This pattern matches if *predicate* applied to the corresponding value is true. If *pat* is present, *pat* must also match.

Match Failure

If no clause matches the value, execution terminates with an error message.

C.4.3 Type Annotations

The form (**:** *type exp*) constrains the type of *exp* to be *type*. If Soft Scheme is able to prove that the type constraint is satisfied, a **:** form has no semantic effect. If Soft Scheme is unable to prove that the type constraint is satisfied, a TYPE check is inserted. This run-time check will fail if it is ever reached.

C.4.4 Declaring Primitives

The form (**define-primitive** *x type*) declares a type for an existing primitive named *x*. The file "custom-chez.ss" uses **define-primitive** to declare types and introduce run-time checks for some of Chez Scheme's non-standard primitive operations.

C.5 Customization

At startup, Soft Scheme loads a system specific customization file that may be used to define non-standard primitive operations. For Chez Scheme, the system specific customization file is named "custom-chez.ss".

Soft Scheme also loads the file ".softschemerc" from the user's home directory, if it exists. Again, this file is usually used to define non-standard primitive operations, in the same manner as the system specific customization file.

C.6 Restrictions on Source Programs

- Names beginning with CHECK- or st: should be avoided.
- The single argument forms of make-vector and make-list cannot be used as they have no sensible type.
- Chez Scheme's **define-structure** with initial values is not handled.
- Not all forms of match patterns supported by the general pattern matching package are handled. The following patterns cannot be used: **...**, **..k**, **not**, **or**, **get!**, **set!**, quasi-patterns, and **?** predicates where the predicate expression is not a primitive or a predicate built by **define-structure**.
- Programs must not rely on the representation of structures as vectors.
- Extend-syntax and defmacro definitions are loaded (by eval) into Scheme and subsequently expanded.
- Code that relies on the order of evaluation of letrec expressions, or that does not order top-level definitions properly (in left-to-right order) is not guaranteed safe.

Bibliography

[1] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems 13*, 2 (April 1991), 237–268. Previously appeared in: *Proc. 16th Annual Symposium on Principles of Programming Languages* (January 1989), 213–227.

[2] Abadi, M., Cardelli, L., Pierce, B., and Rémy, D. Dynamic typing in polymorphic languages. In *Proc. 1992 Workshop on ML and Its Applications* (June 1992).

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

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

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

[6] Standard ML of New Jersey release notes (version 0.93). AT&T Bell Laboratories, November 1993.

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

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

[9] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. The Modula-3 type system. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages* (January 1989), pp. 202–212.

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

[11] Cartwright, R., and Felleisen, M. Extensible denotational language specifications. In *Symposium on Theoretical Aspects of Computer Science* (1994), p. ??

[12] Chambers, C. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992.

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

[14] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[15] Dzeng, H., and Haynes, C. T. Type reconstruction for variable-arity procedures. In *Proc. ACM Conference on Lisp and Functional Programming* (1994), pp. 239–249.

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

[17] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming 17* (1991), 35–75. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

[18] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102* (1992), 235–271. Preliminary version in: Technical Report TR-100, Rice University, June 1989.

[19] Freeman, T. *Refinement Types.* PhD thesis, Carnegie Mellon University, 1993.

[20] Freeman, T., and Pfenning, F. Refinement types for ML. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), 268–277.

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

[22] Greiner, J. Standard ML weak polymorphism can be sound. Tech. Rep. CMU-CS-93-160R, Carnegie Mellon University, September 1993.

[23] Harper, R., and Lillibridge, M. Explicit polymorphism and CPS conversion. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages* (January 1993), pp. 206–219.

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

[25] Henglein, F. Dynamic typing. In *Proceedings of the European Symposium on Programming, LNCS 582* (February 1992), Springer-Verlag, pp. 233–253.

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

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

[28] Hindley, R. J., and Seldin, J. P. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[29] Hoang, M., Mitchell, J., and Viswanathan, R. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science* (June 1993), pp. 15–25.

[30] Hudak, P., et al. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices 27*, 5 (May 1992).

[31] Jagannathan, S., and Weeks, S. Analyzing stores and references in a parallel symbolic language. In *Proc. ACM Conference on Lisp and Functional Programming* (June 1994), pp. 294–305.

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

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

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

[35] Leroy, X. *Typage polymorphe d'un langage algorithmique.* PhD thesis, L'Université Paris 7, 1992.

[36] Leroy, X. Unboxed objects and polymorphic typing. In *Proc. 19th Annual Symposium on Principles of Programming Languages* (January 1992), pp. 177–188.

[37] Leroy, X., and Mauny, M. Dynamics in ML. *Journal of Functional Programming 3*, 4 (1993), 431–463.

[38] Leroy, X., and Weis, P. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), pp. 291–302.

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

[40] MacLachlan, R. A. The Python compiler for CMU Common Lisp. In *Proc. ACM Conference on Lisp and Functional Programming* (1992), pp. 235–246.

[41] MacQueen, D., Plotkin, G., and Sethi, R. An ideal model for recursive polymorphic types. *Information and Control 71* (1986), 95–130. Preliminary version in: *Proc. 11th Annual Symposium on Principles of Programming Languages*, 1984, pp. 165–174.

[42] Martelli, A., and Montanari, U. 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 1976.

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

[44] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

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

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

[47] Paterson, M. S., and Wegman, M. N. Linear unification. *Journal of Computing Systems Science 16*, 2 (April 1978), 158–167.

[48] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[49] Plotkin, G. D. LCF considered as a programming language. *Theoretical Computer Science 5* (1977), 223–255.

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

[51] Rémy, D. Record concatenation for free. Tech. Rep. 1430, INRIA, 1991.

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

[53] Rémy, D. Extension of ML type system with a sorted equational theory on types. Tech. Rep. 1766, INRIA, October 1992.

[54] Shao, Z., and Appel, A. K. Smartest recompilation. In *Proc. 20th Annual Symposium on Principles of Programming Languages* (January 1993), pp. 439–450.

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

[56] Steele Jr., G. L. Building interpreters by composing monads. In *Proc. 21st Annual Symposium on Principles of Programming Languages* (January 1994), pp. 472–492.

[57] Talpin, J.-P., and Jouvelot, P. The type and effect discipline. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science* (June 1992), pp. 162–173.

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

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

[60] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[61] Wright, A. K. Typing references by effect inference. In *Proceedings of the 4th European Symposium on Programming, LNCS 582* (1992), Springer-Verlag, pp. 473–491.

[62] Wright, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation* (1994). To appear in the special issue on State in Programming Languages.

[63] Wright, A. K., and Duba, B. F. Pattern matching for Scheme. Unpublished manuscript, 1993. Available as World Wide Web URL "ftp://cs.rice.edu/public/wright/match.ps.Z".

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