Design and implementation of a reusable type inference engine and its application to Scheme

Brian J. Dorn

TR #05-16
July 2005

Keywords: type inference, Scheme language, Typedscm language
2005 CR Categories:
D.2.13 [*Software Engineering*] Resuable Software — reusable libraries; D.3.2 [*Programming Languages*] Language Classifications — applicative (functional) languages, Scheme; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, data types and structures; D.3.4 [*Programming Languages*] Processors — preprocessors; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — type structure;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

TABLE OF CONTENTS

LIST OF FIGURES . . . iv

ABSTRACT . . . v

CHAPTER 1. INTRODUCTION . . . 1

1.1 Design Overview . . . 2

1.1.1 Problem . . . 2

1.1.2 Goals for Solution . . . 3

1.1.3 Solution Overview . . . 3

1.2 Outline . . . 5

CHAPTER 2. NOTATION . . . 6

2.1 Literate Programming . . . 6

2.2 Variant Record Definitions . . . 7

CHAPTER 3. TYPE HELPER DESIGN AND IMPLEMENTATION . . . 9

3.1 System Specification . . . 9

3.2 Data Structures and Rule Interfaces . . . 10

3.2.1 Interfaces for Type Rules . . . 13

3.3 Rule Processing . . . 14

3.3.1 Base Cases . . . 18

3.3.2 Error Cases . . . 18

3.3.3 Side Procedures . . . 18

3.3.4 Hypothesis Judgements . . . 19

3.3.5 Subtyping Judgements . . . 21

3.4 Summary . . . 22

CHAPTER 4. USER DEFINED COMPONENTS . . . 24

4.1 Type Expressions . . . 24

4.2 Method Dictionary . . . 25

4.3 Annotation Rules . . . 27

4.3.1 Examples of Simple Rule Use . . . 27

4.3.2 Example of Complex Rule Composition . . . 28

4.3.3 Example of Side Definitions with Rule-Seq . . . 28

4.4 Error Handling . . . 30

4.4.1 Error Generation . . . 30

4.4.2 Error Output . . . 31

CHAPTER 5. LANGUAGE AND TYPING EXTENSIONS . . . 32

5.1 Module System . . . 32

5.2 Bidirectional Type Checking . . . 34

5.3 Variant Record Typing . . . 36

CHAPTER 6. IMPLEMENTATION CHALLENGES 37
6.1 Delayed Evaluation 37
6.2 Unification Environment 38
6.3 Runtime Speed 38
CHAPTER 7. FUTURE WORK 40
7.1 DrScheme Integration 40
7.2 Experimental Study 42
CHAPTER 8. CONCLUSION 43
8.1 Related Work 43
8.2 Contributions 44
APPENDIX A. LAMBDA CALCULUS TYPE CHECKER 45
APPENDIX B. APPLICATION DERIVATION COMPLETION 49
BIBLIOGRAPHY 51

LIST OF FIGURES

Figure 1.1 If-Exp Inference Rule in TypedScm00 2
Figure 1.2 Conditional Expression Rule 3
Figure 1.3 TypedScm05 Architecture 4
Figure 1.4 If-Exp Inference Rule in TypedScm05 4

Figure 3.1 Lambda Calculus Grammar 9
Figure 3.2 Lambda Calculus Types 11
Figure 3.3 Lambda Calculus Inference Rules 11
Figure 3.4 Judgement Datatype 12
Figure 3.5 Mixed Datatype 13
Figure 3.6 tc:rule-or Example in TypedScm05 13
Figure 3.7 tc:rule-if Example in TypedScm05 14
Figure 3.8 tc:rule-seq Example in TypedScm05 15
Figure 3.9 Initial Step in Type Derivation 15
Figure 3.10 Outstanding Derivation Parts 16
Figure 3.11 Recursive Derivation on Hypothesis 19
Figure 3.12 TypedScm00 Type Hierarchy [JL96] 21

Figure 4.1 TypedScm05 Type Expressions 24
Figure 4.2 Basic Axiom Use 28
Figure 4.3 Dealing with Lists of Subterms 28
Figure 4.4 Combining Rules using tc:rule-or 29
Figure 4.5 Using Side Definitions to Add Bindings 29
Figure 4.6 Example Error Generation for Conditional 31
Figure 4.7 Example Error: Bad Subtying Relationship on Conditional 31

Figure 5.1 TypedScm05 Module Syntax 33
Figure 5.2 Module for Box ADT 33
Figure 5.3 Error Message in TypedScm00 35
Figure 5.4 Error Message in TypedScm05 35

Figure 7.1 DrScheme and the TypedScm Tool 41
Figure 7.2 DrScheme Type Error Output 41

ABSTRACT

Static type checking allows programmers to locate potential bugs prior to code execution. However, developing a static type checker is a complicated endeavor. Implementers must address a number of concerns including recursion over syntax elements, unification of type variables within environments, and generation of meaningful error messages for users. The inherent complexity of type checkers can lead to code that is difficult to both understand and maintain.

This thesis presents the design and implementation of an abstract type inference engine and its use in the revision of a student-oriented type checker for the Scheme programming language. Our inference engine provides a complete set of unification facilities to programmers for the specification of a type checking system. It allows for a clean separation of unification algorithms, inference rules, and error generation.

We also demonstrate the applicability of the engine by using it to construct a type checker for Scheme targeted at novice programmers. This checker borrows a student-friendly type notation from a previous version and extends its system, providing for language native module support, a more complete treatment of advanced data types, and better error messages.

CHAPTER 1. INTRODUCTION

Learning to program in Scheme is hard. At least some of this difficulty can be attributed to its lack of a static type system. Dynamically-typed languages, like Scheme, provide programmers a great deal of freedom when defining operations over data. They do not unnecessarily restrict programmers to specify particular types for each variable or element of a data structure. By definition, *dynamic languages* do not attempt to validate type correctness at compile time, but rather, they rely on run-time checks to ensure proper data handling. Though there is no explicit notation for types in such languages, typing is still important. Expert programmers have a great deal of intuition about the types associated with their program's data structures and procedures, regardless of dynamic typing mechanisms.

One of the greatest detractors to dynamic typing is that code is inherently difficult to debug [Lin93][WC93][CF91]. This is mainly caused by the fact that errors are not detected until run-time, at which point most systems raise an error condition and execution is aborted. The actual source of the type error may or may not be easily identifiable given the location at which execution terminates.

Static type checking, on the other hand, attempts to identify errors prior to program execution. Typically, static approaches provide more intuitive error messages and aid in locating and correcting errors. However, these systems are more restrictive than those that are dynamic due to the fact that their conservative analysis results in errors for things that only "might" be real problems depending on the run-time context.

Despite the fact that dynamically-typed programming languages have more expressive power than their static counterparts [CF91], the large majority of Scheme programs, and those written by novices in particular, are in fact statically typeable. This is because experience shows that only small portions of typical Scheme programs rely on dynamic typing.

Classroom observations seem to indicate that novice programmers struggle with typing issues when writing code. Misuse of interfaces, improper type casting, and inability to connect type error messages to source code errors are common problems that new programmers make. We believe these problems are amplified by a dynamically-typed language in which students may or may not be confronted with an error, depending on the actual program flow path during runtime. From an educational perspective, static type checking offers an avenue to reinforce good practice in the early stages of learning a language.

The lack of tools for type error discovery traditionally associated with dynamic typing and the impact such tools can have on novice programmers provide motivation for incorporating static type checking features into languages like Scheme in an educational setting.

This thesis presents a variation on a static type checker for Scheme originally created by Jenkins and Leavens [JL96] to aid introductory students at Iowa State University. The original system, hereafter referred to as TypedScm00, includes a student-friendly notation for data types and has been used for several years. Since its original release, ease of maintenance has become increasingly important. However, the nature of its implementation made this inherently difficult. This complexity provided the impetus for a new implementation that expressly addresses the maintenance problem.

The new type checker, referred to as TypedScm05, uses the simple type notation of TypedScm00 but provides a novel implementation approach as well as several enhancements to the original type system. Among the improvements, TypedScm05 is itself designed to be highly extensible. It provides a reusable type checking engine that can be applied to many type checking problems outside of the Scheme language. This engine allows for a separation of concerns within TypedScm05, thereby simplifying the maintenance process. The original polymorphic type system used in TypedScm00 has also been extended to allow for a greater degree of modularity in user code and is more complete in its handling of advanced datatypes.

1.1 Design Overview

Development of a static type checker is by no means a trivial task. These systems must take into account a myriad of concerns including abstract language syntax, type inference rules, unification algorithms, and generated error messages. The inter-relationships between these concerns can be quite complicated. Programmers who write type checkers are, of course, subject to the same difficulties experienced by other software engineers. It is not uncommon for software systems to be designed around the predominant architectural concern while others are forced to take a back seat. Systems which operate on programming languages themselves are influenced strongly by the language's syntax and semantics. However, if other concerns are not also taken into account, the resulting code can become an entangled mix of unification methods, type inference rules, and error generation.

1.1.1 Problem

The previously mentioned TypedScm00 system developed by Jenkins, Clifton, and Leavens is an example of a primarily syntax-driven decomposition. The code in Figure 1.1 provides a tangible, albeit small, example of the problem.

```
(tc:conditional-exp
 (position test consequent alternate)
 (tc:check-for-boolean
  rec-typ-env seq-typ-env unif-env test
  (lambda (ue)
    ((tc:one-must-subtype-other
      "Arms␣of␣if␣expression␣have␣different␣types"
      "Left␣␣arm"
      "Right␣arm")
     rec-typ-env seq-typ-env ue
     consequent alternate result-cont))))
```

Figure 1.1 If-Exp Inference Rule in TypedScm00

This code fragment is responsible for handling the type checking of a conditional expression in Scheme. Essentially it encodes the inference rule[1] given in Figure 1.2. However, the

[1]The notation used for inference rules here is essentially that given by Cardelli in [Car87]. The judgement $\Pi \vdash e : \tau$ means that in a type environment Π, it can be inferred that e has type τ. Within a type environment, $x : \tau$ denotes the binding of variable x to type τ. $\Pi,\ x : \tau$ represents the extension of environment Π by the

underlying rule for any given code section is not obvious to an outside reviewer due to the other concerns dealt with in the same code. Notice that lines 7-9 above contain messages that will be used in error message generation and that there is a good deal of verbosity centered around passing environment parameters (`rec-type-env`, `seq-type-env`, etc).

$$\boxed{\dfrac{\begin{array}{c}\Pi \vdash \mathit{test} : \mathit{boolean} \\ \Pi \vdash \mathit{cons} : t \\ \Pi \vdash \mathit{alt} : s \\ t <> s : \mathit{sub}\ \mathit{super}\end{array}}{\Pi \vdash (\mathit{if}\ \mathit{test}\ \mathit{cons}\ \mathit{alt}) : \mathit{super}}}$$

Figure 1.2 Conditional Expression Rule

Even this small example leaves something to be desired for the programmer responsible for maintaining TypedScm00. The reader can imagine the even greater complexity associated with a less trivial piece of Scheme syntax in this system. Put simply, the problem here is to devise a means of implementation that allows for a cleaner and more maintainable approach to type checking.

1.1.2 Goals for Solution

Simplification of the maintenance process in the type checker serves as our primary motivation for the work that makes up a substantial portion of this paper. Our main goal was to develop a system in which type rules can be easily added or changed independently of other concerns in the system, while at the same time preserving TypedScm00's behavior from the end user's standpoint. In other words, the type system and errors generated needed to be at least as helpful as those that already existed.

1.1.3 Solution Overview

The design of TypedScm05 abstracts many of the previously mentioned concerns into independent modules that come together to form the type checker. Figure 1.3 illustrates the basic architecture.

At the heart of the new system is the type checking engine. This abstract component is responsible for implementing a slightly modified Hindley-Milner unification algorithm [Mil78] [Car87]. Internally this engine is oblivious to the representation of syntax, types, and errors; however, it does make use of abstract procedures that must be provided through a *method dictionary* in order to manipulate the specific representations. The method dictionary is a data structure which encapsulates several required procedures defined outside of the scope of the type checking engine. This engine is therefore reusable and may be applied to various forms of type checking problems, limited only by the components which are "plugged in" through the dictionary.

The external modules (shown in grey in the diagram) are required to provide a basic set of procedures in order to work with the type checking engine, but they typically implement

binding $x : \tau$. The notation $t <> s$: *subtype suptype* is used to specify that either t is a subtype of s or s is a subtype of t, where *subtype* and *suptype* are aliases for the appropriate values on the left-hand side. Clauses appearing above the horizontal bar are said to imply those below it.

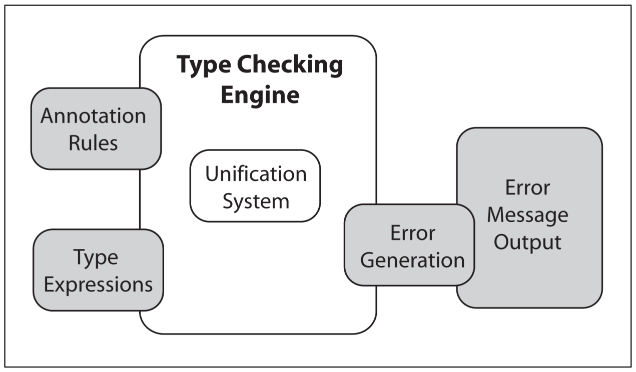


Figure 1.3 TypedScm05 Architecture

a great deal of functionality within themselves related to their specific function. Depending on the application, the programmer may specify an optional error output mechanism. Given that this component is entirely separated from type checking, it is possible that a programmer could specify multiple output methods to customize the system to a particular environment.

Perhaps the most important aspect of the type checking engine is its ability to separate rule definitions from the other concerns. Not only are these rules no longer tangled with error message text and unification, but they are also defined using a syntax that directly corresponds to a mathematical notation, like that shown in Figure 1.2. The engine provides a type checker implementer a set of procedures which are used to specify a rule. For comparison, the Figure 1.4 illustrates the code for the same conditional expression shown earlier. Here lines 8-14 correspond directly to the mathematical way in which we specify the same rule, with the minor exception that the ordering of the syntax is slightly modified (prefix operators :- and : versus infix operators $\vdash$ and :).

```
   (tc:conditional-exp
    (position test consequent alternate)
    (let* ((t1 (tc:new-variable-type-expr))
           (t2 (tc:new-variable-type-expr))
 5         (supertype-var (tc:new-logicalvar))
           (subtype-var (tc:new-logicalvar))
           (supertype (tc:variable-type-expr supertype-var)))
       (tc:rule (list
                 (:- pi (: test *tc:boolean*))
10               (:- pi (: consequent t1))
                 (:- pi (: alternate t2))
                 (<:> t2 t1 subtype-var supertype-var))
                ;; ----------------------------------------
                (:- pi (: e supertype)))))
```

Figure 1.4 If-Exp Inference Rule in TypedScm05

In addition to the simple rule shown here, the type checking engine allows programmers to make use of side conditions and side definitions with in their rules. These features are outlined in more detail in chapter 4.

TypedScm05 utilizes this engine for its implementation, but also makes several other improvements beyond those impacting readability and maintainability. This work introduces a module system that is a typeable subset of modules in MzScheme [FFF$^+$97]; MzScheme's module system allows users to specify a module that only uses names from modules it explicitly imports. This feature allows for independent type checking of modules and was not available in TypedScm00.

Where possible, TypedScm05 also makes use of more advanced typing rules in an effort to provide better error messages to end users. Specifically we implement a bidirectional inference method like that described in [PT98a]. The ability to switch from traditional bottom-up inference to top-down checking when a previous type definition exists allows a greater degree of accuracy for generating error messages. The top-down approach enables the type checker to point users to a particular expression within a procedure body that is suspected to be the cause of the error, rather than simply reporting a mismatch between a procedure's declared and inferred types as in TypedScm00. Additionally, this implementation provides insight into the power of the type checking engine, as it is able to handle features like bidirectionality when given the appropriate rules.

1.2 Outline

Throughout this paper, it is assumed that the reader has a functional knowledge of basic type checking and is literate in the Scheme programming language. The remainder of this thesis proceeds as follows. Necessary notation is introduced in chapter 2. Chapter 3 highlights important aspects of the type checking engine implementation. Use of the engine is discussed in chapter 4 through example components taken from our Scheme system. Chapter 5 examines new additions to the original type system and their impacts on the system's overall usability. A brief overview of notable implementation challenges is given in chapter 6 followed by a discussion of future work in chapter 7. Lastly, chapter 8 concludes.

CHAPTER 2. NOTATION

We begin by presenting a brief introduction to the notation used throughout this paper. A style of coding and documentation known as *literate programming* is defined in section 2.1. We elaborate on a variant record syntax for Scheme used in later code examples in section 2.2.

2.1 Literate Programming

At various points in the explanation of our system it is necessary to examine source code from the actual implementation. These sections consist of a mix of Scheme source code and exposition. In an effort to keep code segments and their comments to a digestible length for the reader, we adopt a notation similar to that used in literate programming.

The idea of literate programming was introduced in the mid-1980s by Knuth as a way to interleave source code and descriptive text into a single document [Knu92]. His work frees authors from trying to explain programs in the order required by a compiler and allows them to choose an order that more naturally fits how a program actually operates.

In addition to notation, Knuth developed a tool called Web that can manipulate literate programs for either compilation or display. His original system uses the Pascal language, but the approach is applicable to any programming language. In fact, similar tools have been developed for nearly every language imaginable [Ram94].

Though TypedScm05 is not implemented using one of the many literate programming tools, Knuth's notation will be used here for descriptive simplicity. His notation defines the notion of a code fragment or *chunk*. These chunks can appear in any order, interleaved with textual information, and may internally reference other chunks. The following is an example code chunk:

$<$*FindDuplicates* 2.1$>$ $\equiv$

```
(define tc:find-duplicates
  (letrec ((look-for-dups
           <LookforDuplicates 2.1> ))
    (lambda (ls)
      (look-for-dups ls '()))))
```

Each code chunk is given a unique name made up of an identifier and a section number (e.g., $<$*FindDuplicates* 2.1$>$). Section numbers refer to this document, rather than the files containing the actual code. The names are used to refer to chunks which appear elsewhere in the text. Typically, code that needs more detailed description is abstracted into its own chunk. In the above program the reference to $<$*LookforDuplicates* 2.1$>$ illustrates this use; its definition appears on the next page.

$<$ *LookforDuplicates* 2.1 $>\equiv$

```
(lambda (ls dups)
  (cond
   ((null? ls) dups)
   ((member (car ls) (cdr ls))
    (look-for-dups (cdr ls) (cons (car ls) dups)))
   (else (look-for-dups (cdr ls) dups))))
```

Thus, the meaning of $<$ *FindDuplicates* 2.1 $>$ in the actual source program combines all references to internal chunks in one procedure:

```
(define tc:find-duplicates
  (letrec ((look-for-dups
            (lambda (ls dups)
              (cond
               ((null? ls) dups)
               ((member (car ls) (cdr ls))
                (look-for-dups (cdr ls) (cons (car ls) dups)))
               (else (look-for-dups (cdr ls) dups)))) ))
    (lambda (ls)
      (look-for-dups ls '()))))
```

2.2 Variant Record Definitions

On several occasions, this paper and the code to which it refers requires the creation of special datatypes within Scheme. These situations are treated using variant record definitions presented with the `define-datatype` syntax specified in *Essentials of Programming Languages (second edition)*[FWH01]. *Variant records* are data structures that allow a particular record instance to take on one several possible forms, called *variants*. For example, suppose we want to represent a binary tree with its own datatype.

A *binary tree* is a tree that is either empty or a root node with two descendant nodes, left and right, which are also binary trees. Here we also allow a number to be associated with each node in the tree. The Scheme code below defines a type, `bintree`, for a binary tree record and its two possible variants using the `define-datatype` syntax.

```
(define-datatype bintree bintree?
  (empty-bintree)
  (root-node
    (num number?)
    (left bintree?)
    (right bintree?)))
```

This syntax also defines a type predicate, `bintree?`, and constructors for each variant case, `empty-bintree` and `root-node`. Thus, we can create and test for `bintrees` as follows:

```
(bintree? 8675309) ==> #f
(bintree? (empty-bintree)) ==> #t
(bintree? (root-node 5 (empty-bintree) (empty-bintree))) ==>  #t
(bintree? (root-node 1 (root-node 2 (empty-bintree) (empty-bintree))
                       (root-node 3 (empty-bintree) (empty-bintree)))) ==> #t
```

Variants are manipulated using a new expression called `cases`. The following code illustrates how to define a procedure that computes the sum of all node values in a tree.

```
(define sumtree
  (lambda (tree)
    (cases bintree tree
           (empty-bintree () 0)
           (root-node (num left right)
             (+ num (sumtree left) (sumtree right))))))
```

CHAPTER 3. TYPE HELPER DESIGN AND IMPLEMENTATION

The abstract type checking engine described in chapter 1 is a collection of Scheme modules which provide type checking functionality to client code. The implementation strategy is taken from an unpublished prototype for such a system written in Haskell by Leavens, but has been enhanced to support additional functionality like subtyping within an implementer's type system. (The original system attempted to model ideas from Schmidt's *The Structure of Typed Programming Languages* [Sch94].) The aspects of interest to the reader here are collectively called the *type helpers*. Procedures making up the type helpers are responsible for processing type annotation rules and define the interface used by implementers to specify a complete type checker.

This chapter highlights the design and implementation of the type helpers. We begin with a high-level specification of the system in section 3.1, followed by a discussion of our representation for data and rules in section 3.2. A detailed description of how inference rules are processed is given in section 3.3. The chapter is summarized in section 3.4.

3.1 System Specification

Before we examine the implementation details of the type helpers, it is useful to get a better picture of the general goal. In this section, we examine a design approach for specifying an automated type checker for Church and Curry's *lambda calculus* with constants [CFC58]. The aim is to implement a type checker using Scheme that can process the lambda calculus language and infer types using Hindley-Milner inference rules [Mil78]. Before a discussion of type checking is possible, it is necessary to first define the language on which we will operate.

A grammar for the lambda calculus is given in Figure 3.1. The formal concrete syntax is shown on the right of the figure and a corresponding Scheme variant record definition appears on the left. This Scheme datatype will serve as the abstract syntax for the type checker.

Scheme Representation **λ Calculus**

```
(define-datatype tc:lambda-calc tc:lambda-calc?
  (tc:le-self-evaluating
   (datum datum?))
  (tc:le-varref
   (variable symbol?))
  (tc:le-procedure-call
   (operator tc:lambda-calc?) (operand tc:lambda-calc?))
  (tc:le-lambda-exp
   (formal symbol?) (body tc:lambda-calc?)))
```

$$
\begin{aligned}
\langle e \rangle ::=\ & se \\
 \mid\ & x \\
 \mid\ & \langle e_0 \rangle\ \langle e_1 \rangle \\
 \mid\ & \lambda x.\ \langle e \rangle
\end{aligned}
$$

Figure 3.1 Lambda Calculus Grammar

With a grammar for syntax defined, the next step in developing a type checker is to specify a representation for the types associated with data in the language. Figure 3.2 on the following page depicts type expressions within the system (where a basic type, $B \in \{number, symbol, string, char, boolean\}$). Note that the final two type expression variants in the Scheme code are outside those defined by the formal syntax. Polymorphic type variables are represented with `tc:variable-type-expr`, and errors are detected with `tc:error-type-expr`. These special types will allow rules to be type checked and are discussed in greater detail in subsequent chapters.

With these notions in place, it is possible to discuss type checking rules. Ideally, a type checker's implementation would closely mirror the inference rules defined for the language. Thus, we propose a system that is capable of directly evaluating type rules. Figure 3.3 illustrates our implementation style versus the formal notation of the lambda calculus.

In the code appearing on the left, we define the manner in which each of the lambda calculus expression variants are checked. The free variable `e` is used to represent the current syntax element being checked (as if this entire figure were specifying the clauses of `(cases tc:lambda-calc e ...)`). The forms `tc:axiom`, `tc:rule`, and `tc:rule-if` are introduced to model the axioms, rules, and rules with side conditions in the formal specification. In each case, there is a direct correspondence between the Scheme and formal representations.

Lines 1-5 of the code define the axiom used to assign a type to a self-evaluating expression. Specifically, line 3 looks up the associated basic type, B, which is used in the axiom definition on line 5. Variable references require the use of a rule with a side condition in order to specify that $x : \tau \in \Pi$. Such variable references are handled in the clause given on lines 6-16. The side condition that the identifier name, `variable`, is bound in the environment, `pi`, appears on lines 11-12. The side definition that follows on lines 13-14 allows the conclusion to refer to the associated type value found in `pi`.

More involved rules for applications and lambda expressions are shown on lines 17-26 and lines 27-35, respectively. In the application variant case, `tc:le-procedure-call`, we begin by declaring two new type variables, `rt` and `operand-type`, that correspond to the variables τ' and τ in the formal rule. A simple rule is written with `tc:rule` that encodes the two hypotheses using these type variables and the identifiers `operator` for e_0 and `operand` for e_1. Similarly, the clause for lambda expressions (lines 27-35) is translated using `tc:rule` and the appropriate type variables. Extension of type environment Π in the hypothesis is modeled with a call to `tc:extend-env` seen on line 32.

Clearly, the notation presented here is analogous to the type rules themselves and can easily be understood. Given an infrastructure capable of processing the somewhat "magical" `tc:rule` and `tc:rule-if`, it is entirely possible to treat type checker implementation in this manner. Just as people manipulate type checking rules to perform derivation proofs, one could imagine a system that automatically creates proof trees from these rule definitions. In fact, the type helpers mentioned in the introduction to this chapter do just that. The remainder of this chapter explores exactly how this process takes place within our type checking engine.

3.2 Data Structures and Rule Interfaces

Before looking at processing, we introduce the type helper interfaces. *Logical variables* are the most basic element used in the type checker (denoted `tc:logicalvar` in our system). We define a logical variable here as a variable bound to a specific value, another logical variable,

Scheme Representation **λ Calculus**

```
(define-datatype tc:type-expr tc:type-expr?
  (tc:basic-type-expr (symbol symbol?))
  (tc:function-type-expr
   (arg-type tc:type-expr?) (result-type tc:type-expr?))
  ;; For type helpers
  (tc:variable-type-expr (lvar tc:logicalvar?))
  (tc:error-type-expr))
```

$$\begin{aligned} \langle\tau\rangle \ ::=& \\ & B \\ \mid & \ \langle\tau_0\rangle \rightarrow \langle\tau_1\rangle \end{aligned}$$

Figure 3.2 Lambda Calculus Types

Scheme Representation **λ Calculus**

```
(tc:le-self-evaluating
  (datum)
  (let ((B (tc:infer-simple-datum-type datum)))
    ;; ---------------------------------------
    (tc:axiom (:- pi (: e B)))))
```

$$\frac{}{\Pi \vdash b : B}$$

```
(tc:le-varref
 (variable)
 (let ((lv (tc:new-logicalvar)))
   (tc:rule-if
    '() ;; no hypotheses here
    (lambda (ts)
      (tc:env-bound? variable pi))
    (lambda (ts)
      (tc:type-expr-bind lv (tc:env-value variable pi)))
    ;; ----------------------------------------------
    (:- pi (: e (tc:variable-type-expr lv))))))
```

$$\frac{x : \tau \in \Pi}{\Pi \vdash x : \tau}$$

```
(tc:le-procedure-call
 (operator operand)
 (let ((rt           (tc:new-variable-type-expr))
       (operand-type (tc:new-variable-type-expr)))
   (tc:rule (list
            (:- pi (: operator
                      (tc:function-type-expr operand-type rt)))
            (:- pi (: operand operand-type)))
         ;;-----------------------------------------------
         (:- pi (: e rt)))))
```

$$\frac{\Pi \vdash e_0 : \tau \rightarrow \tau' \qquad \Pi \vdash e_1 : \tau}{\Pi \vdash e_0\ e_1 : \tau'}$$

```
(tc:le-lambda-exp
 (formal body)
 (let ((rt (tc:new-variable-type-expr))
       (formal-type (tc:new-variable-type-expr)))
   (tc:rule (list
            (:- (tc:extend-env pi formal formal-type)
                (: body rt)))
         ;; --------------------------------------------------
         (:- pi (: e (tc:function-type-expr formal-type rt))))))
```

$$\frac{\Pi, x : \tau \vdash e : \tau'}{\Pi \vdash (\lambda x : \tau.\ e) : \tau \rightarrow \tau'}$$

Figure 3.3 Lambda Calculus Inference Rules

or nothing at all in the unification system's environment. They are used during unification to gather typing constraints on various pieces of syntax.

Additionally, there are three primary data structures used by the type helpers in the processing of rules: attributed syntax pairs, typing judgements, and mixed hypothesis elements. Of these, syntax pairs are the most straightforward. *Attributed syntax pairs* are simply a pair that contains a piece of syntax and some attribute. We provide a procedure named "`:`" as the constructor for such pairs. For our concerns, the attribute consists of a type expression, and these pairs associate the given type information with a particular piece of syntax.

By themselves, these syntax pairs are not very interesting. Things become more complex when they are incorporated into typing judgements. A *judgement* is used to specify a typing constraint to be checked by the system. The code in Figure 3.4 defines a variant record for judgements. Each variant represents a different type of judgement in TypedScm05.

```
(define-datatype tc:judgement tc:judgement?
  (:-d (env tc:environment?) (attr-pair (tc:attrib-pair-of datum? datum?)))
  (:?- (lvar tc:logicalvar?) (attr-pair (tc:attrib-pair-of datum? datum?)))
  (<:  (subtype datum?) (supertype datum?))
  (<:> (t1 datum?) (t2 datum?)
       (subtype-var tc:logicalvar?) (supertype-var tc:logicalvar?)))
```

Figure 3.4 Judgement Datatype

We provide four different forms of typing judgements. The first, `:-d`, is the most common type of judgement and corresponds to a simple assertion that in environment `env` a given attributed syntax pair, `(: syn type)`, holds[2],[3]. The second form, `:?-`, allows for the environment field to be a logical variable which can be defined elsewhere. This is only useful in certain declaration forms. The last two judgement forms allow for the specification of subtyping relationships in the system. In most cases the simple subtype assertion form, `<:`, is sufficient; however, we also provide a more general form, `<:>`. This form is used to specify that one of two types, `t1` and `t2` must subtype the other. It also allows for two logical variables that will be bound to the appropriate types if the relationship holds. It is made available primarily as a convenience to the rule programmer as it is possible to convert it into a composition of two or more rules using the simple subtype judgement.

These data structures are adequate for most judgements a programmer would want to encode. However, an additional structure is needed in the case where a programmer needs to mix true hypotheses with side conditions and side definitions that are to be executed to either alter the unification environment or to test for some correctness condition before proceeding to a subsequent hypothesis. The variant record shown in Figure 3.5 defines the mixed datatype used in these cases.

The `tc:mixed` type provides three different possibilities. The first, `tc:hyp`, is a wrapper for the standard judgements described above. The `tc:when` variant is used to specify side conditions, and `tc:def` allows for incorporation of side definitions. Examples of how each of these used is specified in greater detail in chapter 4.

[2]Examples shown thus far have used the notation `:-` for this simple judgement case. While `:-d` and `:-` are technically different things, the reader may consider them aliases for one another for the moment. The need for two separate operators is discussed later in section 6.1.

[3]The term *holds* in this sense means that the relationship denoted by the syntax pair is valid within the given environment context.

```
(define-datatype tc:mixed tc:mixed?
  (tc:hyp (judgement tc:judgement?))
  (tc:when (f (-> (list-of datum?) boolean?)))
  (tc:def (f (-> (list-of datum?) (tc:subst-of datum?)))))
```

Figure 3.5 Mixed Datatype

3.2.1 Interfaces for Type Rules

These datatypes are used in conjunction with one of five procedures defined by the type helpers to specify a type rule. The given procedures are: `tc:axiom`, `tc:rule`, `tc:rule-or`, `tc:rule-if`, and `tc:rule-seq`. The axiom procedure, `tc:axiom`, is used to specify a rule that has no hypotheses; the judgement provided in an axiom is always treated as true in the type checker. The `tc:le-self-evaluating` rule of the lambda calculus in Figure 3.3 exemplifies the use of `tc:axiom`.

The most common rule helper is `tc:rule`, and we have already seen several examples of it in Figures 1.4 and 3.3. Using this helper, implementers specify a list of judgements which serve as hypotheses and conclusion judgement which is true if and only if no type errors are found while processing the hypotheses.

More advanced type rules are specified using the other three type helpers. A convenience that allows a more fine-grained splitting of rules into cases is the `tc:rule-or` combinator. It combines two or more rules and checks that at least one of them holds.

An example rule for a hypothetical equals expression is shown in Figure 3.6. This rule asserts that a `tc:equals-exp` expression is valid with a boolean type if its two subforms, `e1` and `e2`, have either both number or both boolean types.

```
(tc:equals-exp
 (e1 e2)
 (tc:rule-or
  (tc:rule (list
             (:- pi (: e1 *tc:number*))
             (:- pi (: e2 *tc:number*)))
            ;; ------------------------
            (:- pi (: e *tc:boolean*)))
  (tc:rule (list
             (:- pi (: e1 *tc:boolean*))
             (:- pi (: e2 *tc:boolean*)))
            ;; -------------------------
            (:- pi (:e *tc:boolean*)))))
```

Figure 3.6 tc:rule-or Example in TypedScm05

Rules with simple side conditions can be specified using the `tc:rule-if` form. It requires a list of hypotheses and a conclusion like standard rules, but it also takes two procedures that define side conditions. The first is a function from type attributes to a boolean value. If this returns true, then the type attributes are passed to the second procedure which can return a substitution defining any number of new variables in the unification environment. When no definitions need to be made, the second procedure returns the null substitution.

Figure 3.7 depicts a simple use of this rule form that checks a lambda expression with multiple formal parameters. First, the `body` is checked in an environment containing bindings for the `formals`, as in the lambda calculus. The side condition enforces the restriction that the formal parameters must be unique, and the side definition is not used.

```
(tc:lambda-exp
 (formals body)
 (let ((rt (tc:new-variable-type-expr))
       (formal-ts (tc:listof-new-variable-type-expr (length formals))))
   (tc:rule-if (list
              (:- (tc:extend-mult-env pi formals formal-ts)
                  (: body rt)))
             (lambda (ts)                    ;; Side Condition
               (tc:no-duplicates? formals))
             (lambda (ts)                    ;; Side Definition
               (tc:type-expr-null-subst))
             ;; ----------------------------------------------
             (:- pi (: e (tc:function-type-expr formal-ts rt))))))
```

Figure 3.7 tc:rule-if Example in TypedScm05

The only thing one cannot accomplish with the above type helpers are rules that, like sequential declarations, need to alter the environment in the middle of the rule. For this we provide a specialized form, `tc:rule-seq`. This helper requires a list of `tc:mixed` data elements and a conclusion which is a normal judgement. Rather than the independent processing of hypotheses present in the other helpers, `tc:rule-seq` processes its argument list sequentially. The list may contain hypotheses, side conditions, and side definitions in any order. Elements tagged with `tc:def` or `tc:when` are passed the list of attributes computed thus far. Ones marked as side definitions can define new values for logical variables to be used later, and those which are side conditions are tests which are used to terminate type checking when they return false.

A sample appears in Figure 3.8 for a simple sequential declaration form, `tc:then-exp`, is checked here. If the first declaration, `d1`, type checks, its bindings are added to original environment to check the second declaration, `d2`. When both declarations successfully check, their bindings are combined and provided in a declaration type used as the conclusion's inferred type.

These datatypes and type-helping procedures provide the functionality needed by an external programmer to specify how type checking should proceed in their specific application.

3.3 Rule Processing

Each of the five type helpers supplies a slightly different feature set to programmers. However, it is not necessary to discuss each of them individually due to the fact that `tc:rule` and `tc:rule-if` can be converted into special instances of the more general `tc:rule-seq` form. These translations are quite straightforward and merely involve wrapping up existing judgements and side procedures with the corresponding `tc:mixed` variants. For this reason, we limit the remaining discussion to `tc:rule-seq`.

The processing of sequential rules (and thus all rules) proceeds in a manner that is analogous to standard type derivation. The item of syntax being checked serves as the starting point for type inference. For example, assume we would like to check the application:

$$(\lambda x.\ x)\ n \text{ in environment } \Pi = \{n : \mathit{number}\}$$

```
(tc:then-exp
 (d1 d2)
 (let ((pi1 (tc:new-variable-type-expr))
       (pi2 (tc:new-variable-type-expr))
       (pi3 (tc:new-logicalvar))
       (pi4 (tc:new-logicalvar)))
   (tc:rule-seq
    (list
     (tc:hyp  (:- pi (: d1 pi1)))
     (tc:when (lambda (ts) (tc:all tc:declaration-type-expr? ts)))
     (tc:def  (lambda (ts) (tc:type-expr-bind
                            pi3
                            (tc:declaration-type-expr
                             (tc:env-join
                              pi (tc:declaration-type-expr->env (car ts)))))))
     (tc:hyp  (:?- pi3 (: d2 pi2)))
     (tc:when (lambda (ts) (tc:all tc:declaration-type-expr? ts)))
     (tc:def  (lambda (ts) (tc:type-expr-bind
                            pi4
                            (tc:declaration-type-expr
                             (tc:env-join
                              (car ts)
                              (cadr ts)))))))
    ;;------------------------------------------------------------------
    (:- pi (: e (tc:variable-type-expr pi4))))))
```

Figure 3.8 tc:rule-seq Example in TypedScm05

Figure 3.9 shows the first step of derivation. Based on the type of syntax (an application in this case) we choose the correct rule. At this stage, we assign type variables to the elements since we do not know anything more specific.

$$\frac{\Pi \vdash (\lambda x : \tau_1 .\ x) : \tau_3 \rightarrow \tau_2 \qquad \Pi \vdash n : \tau_3}{\Pi \vdash (\lambda x : \tau_1 .\ x)\ n : \tau_2}$$

Figure 3.9 Initial Step in Type Derivation

This step is analogous the invocation of `tc:rule` given the arguments as specified in Figure 3.3. Verifying that the application is correctly typed requires us to examine each of the hypotheses for the sub-expressions and ensure no errors exist in them. All rule processing begins in a similar fashion and is depicted in <*RuleSeqHelper* 3.3>.

As with manual derivations, we must process the sub-expressions based on the hypotheses given in the rule definition. Here, we invoke a helping procedure to validate the list of `tc:mixed` hypotheses, `hs` (line 8). If no type errors result during the processing of this list, we generate a new attributed syntax pair that associates the syntax item and type given in the original conclusion for the rule (lines 14-16). The system also adds type information gleaned during hypothesis checking[4] which can be used at a later point in error processing. If errors occur, the resulting syntax pair contains an error type instead of the type given in the conclusion. Here we make use of one of the procedures from the method dictionary, `gen-error-rule`, allowing complete flexibility for the handling of errors which occur during rule processing (line 19).

[4]This information forms a syntax tree with inferred type annotations for each syntactic element.

$<$*RuleSeqHelper* 3.3$> \equiv$

```
(define tc:rule-seq-helper
  (lambda (md annotate)
    <UnpackMethodDictionary 4.2>
    (lambda (hs conc)
      (cases tc:judgement conc
             (:-d (env attr-pair)
                  (let ((result_trees
                         ((tc:sequence md annotate) hs '() ))
                        (syn (tc:attrib-pair->syn attr-pair))
                        (tau (tc:attrib-pair->type attr-pair)))
                    (if (not (tc:error-noted?))
                        (let* ((s (tc:get-current-subst))
                               (ts (tc:map-tops (app s) result_trees)))
                          (: syn (tc:node ((app s)
                                           (tc:force-if-promise tau))
                                          ts)))
                        ;; Type error
                        (: syn (tc:node
                                (gen-error-rule syn result_trees)
                                result_trees)))))
             (else (error "Conclusion␣judgement␣must␣be␣a␣:-␣judgement")))
      )))))
```

In reality only a small portion of the type inference work (that relating to conclusions) pertains to this code. The majority of the complexity comes in processing the sub-expressions and checking their mutual constraints. Revisiting the same procedure application example, Figure 3.10 illustrates the outstanding areas in the derivation with vertical ellipses.

$$\dfrac{\dfrac{\boxed{\;\vdots\;}}{\Pi \vdash (\lambda x : \tau_1.\ x) : \tau_3 \to \tau_2} \qquad \dfrac{\boxed{\;\vdots\;}}{\Pi \vdash n : \tau_3}}{\Pi \vdash (\lambda x : \tau_1.\ x)\ n : \tau_2}$$

Figure 3.10 Outstanding Derivation Parts

The missing parts of the derivation must be filled in by further annotating each sub-expression and its hypothesis in turn. Our internal helping procedure for processing lists of hypothesis and optional side procedures appears above in $<$ *tc* : *sequence* 3.3 $>$. The code may appear daunting at first, but the general idea behind `tc:sequence` is quite simple. If the list of terms passed as an argument, `hs`, contains at least one element, we determine how to proceed based on a case analysis of the first element of `hs`. The kind of `tc:mixed` element we are considering leads to an appropriate action. Remember that, in addition to hypotheses, our lists may contain optional side procedures that must be handled specially. Furthermore the typical `tc:hyp` hypothesis elements, must be separated by their type of judgement. Thus any one of six normal actions will be taken; error checking from previous hypothesis adds another four cases as seen on lines 17, 22, 29, and 33.

The subsections that follow define the specific behaviors for each of the six cases referred to within $<$ *tc* : *sequence* 3.3 $>$. Subsection 3.3.1 cover the base case, and subsection 3.3.2 reviews the error cases. Implementation for side conditions and side definitions is outlined in subsection 3.3.3. The judgements containing environments are handled by subsection 3.3.4, and those pertaining to subtyping are dealt with in subsection 3.3.5.

<tc:sequence 3.3> ≡

```
(define tc:sequence
  (lambda (md annotate)
    <UnpackMethodDictionary 4.2>
    (lambda (hs attrs)
      (letrec ((recurse (tc:sequence md annotate))
               (vars-in (tc:vars-in md)))
        (let ((check-subtype-judgement
               (tc:check-subtype-judgement hs md app bind attrs recurse)))
          (if (null? hs)
              <BaseCase 3.3.1>
              (cases tc:mixed
                (tc:force-if-promise (car hs))
                (tc:def (f)
                        (if (tc:error-noted?)
                            <ContinueCase 3.3.2>
                            <SideDefinitionCase 3.3.3> ))
                (tc:when (p)
                         (if (tc:error-noted?)
                             <ContinueCase 3.3.2>
                             <SideConditionCase 3.3.3> ))
                (tc:hyp
                 (judgement)
                 (cases tc:judgement judgement
                        (:-d (env attr-pair)
                             (if (tc:error-noted?)
                                 <ContinueCase 3.3.2>
                                 <NormalHypothesisCase 3.3.4> ))
                        (:?- (lvar attr-pair)
                             (if (tc:error-noted?)
                                 <ContinueCase 3.3.2>
                                 <VariableEnvironmentHypothesisCase 3.3.4> ))
                        (<: (subtype supertype)
                            <SimpleSubtypeCase 3.3.5> )
                        (<:> (t1 t2 subtype-var supertype-var)
                             <EitherSubtypeCase 3.3.5> )))
                )))))))))
```

3.3.1 Base Cases

The most simple case is the base case, which executes when the list of hypotheses contains no elements. Its implementation is trivial; we simply return the list of type information gathered for the sub-expressions, `attrs`, accumulated earlier. This is a list of annotated syntax trees with one tree for each sub-expression.

<*BaseCase* 3.3.1> ≡

```
attrs
```

3.3.2 Error Cases

Another relatively simple case also requires little additional processing. This case primarily arises when a type error has already been encountered in a previous judgement. Rather than processing anything further at this stage, we just continue to the next judgement. The current one is effectively ignored.

<*ContinueCase* 3.3.2> ≡

```
(recurse (cdr hs) attrs)
```

3.3.3 Side Procedures

As mentioned earlier in this chapter, there are two different types of side procedures: those that are conditions and others that provide additional definitions. Side conditions are used to incorporate additional constraint checking into a rule. For example, one could use them to require that no duplicate identifiers occur in a list of formal parameters. Evaluation of a condition, shown in < *SideConditionCase* 3.3.3 >, involves calling the given procedure, denoted `p` in line 1, with the type information gathered thus far. If this condition holds and results in a true value, unification continues normally. Unsuccessful tests of the side condition are noted as errors prior to continuation (line 5).

<*SideConditionCase* 3.3.3> ≡

```
  (if (p (map tc:root attrs))
      (recurse (cdr hs) attrs)
      (begin
        ;; Update unification env with error
5       (tc:note-error!)
        <ContinueCase 3.3.2>))
```

Side definitions are processed in a manner similar to their conditional counterparts by <*SideDefinitionCase* 3.3.3>. We first begin by evaluating the given procedure, denoted below by `f`. Its result is a substitution which provides additional definitions for logical variables that are to be added to the current unification environment. This is accomplished by first composing additional definitions with those already present and then updating the global environment accordingly (lines 3-7). Evaluation then proceeds with the next element in the list.

<*SideDefinitionCase* 3.3.3> ≡

```
(let ((delta_subst (f (map tc:root attrs))))
;; Update the current environment
  (tc:set-current-subst-list!
    (list
      (tc:subst-compose
        (tc:get-current-subst)
        delta_subst)))
  (recurse (cdr hs) attrs))
```

3.3.4 Hypothesis Judgements

The most interesting of the cases is the one that processes a normal hypothesis. To verify the first hypothesis in our sample derivation, we must recursively annotate its syntax and check that our previously guessed expected type and the inferred type unify. Figure 3.11 shows this process[5]. Similarly, the code in <*NormalHypothesisCase* 3.3.4> performs this task.

$$\dfrac{\dfrac{\dfrac{x:\tau_4 \in \Pi, x:\tau_4}{\Pi, x:\tau_4 \vdash x : \genfrac{}{}{0pt}{}{\tau_4}{\tau_5}}}{\Pi \vdash (\lambda x : \genfrac{}{}{0pt}{}{\tau_4}{\tau_1} .\ x) : \genfrac{}{}{0pt}{}{\tau_4 \to \tau_5}{\tau_3 \to \tau_2}} \qquad \dfrac{\boxed{\vdots}}{\Pi \vdash n : \tau_3}}{\Pi \vdash (\lambda x : \tau_1 .\ x)\ n : \tau_2}$$

Figure 3.11 Recursive Derivation on Hypothesis

Processing this form begins by annotating the inner piece of syntax with its inferred type and determining whether an error occurs on the subform (lines 1-6). We then resolve any known type variables in the inferred type and extract the expected type from the original judgement form (lines 9-15). The expected and inferred types are then adjusted to make sure that they have mutually exclusive type variables (lines 16-21).

The unification algorithm, `tc:mgu`, is invoked on line 22 in an attempt to compute a most general unifier for the expected and inferred types. If the types can not be unified we note that an error has occurred and place a type mismatch error into the annotation list (lines 24-30). It is important to note that this is the sole location of error type generation in the sequence routine. The only errors which occur at this level are simple mismatches between given and actual types. This does not, however, limit the error notification capabilities of the system. A greater degree of error information can be obtained when `tc:sequence` returns to its caller, `tc:rule-seq-helper`, and context information for an entire rule can be analyzed by the user-defined procedure `gen-rule-error`.

When no unification error is detected, the resulting substitution is added to the current unification environment (lines 32-36) and the remaining hypothesis are processed with the newly inferred type added to the accumulation list (lines 37-40).

The other hypothesis case, <*VariableEnviornmentHypothesisCase* 3.3.4>, dealing with variable environments involves much less work. Judgements of this form have a logical variable instead of a concrete environment. Thus, processing begins by attempting to resolve the

[5]In this diagram we show all type variables used as per the rules. If two variables appear in a vertical column, the variable on top is the inferred type from the upper judgements, and the one on the bottom is the expected type specified by the conclusion below.

$<NormalHypothesisCase\ 3.3.4> \equiv$

```
(let* ((syn (tc:attrib-pair->syn attr-pair))
       (recur-result (annotate env syn)))
  (if (tc:error-noted?)
      ;; Unification error on sub annotation
      (recurse (cdr hs)
               (append attrs (list (tc:attrib-pair->type recur-result))))
      ;; So far so good...
      (let* ((result-attr (tc:attrib-pair->type recur-result))
             (attr
               (tc:node
                 ((app (tc:get-current-subst)) (tc:root result-attr))
               (tc:leaves result-attr)))
             (expected
              ((app (tc:get-current-subst))
               (tc:force-if-promise (tc:attrib-pair->type attr-pair)))) )
        (let* ((renaming
                (if (null? (tc:intersect (vars-in (tc:root attr)) (vars-in expected)))
                    (lambda (trm) trm)
                    ((tc:rename-vars md)
                     (+ 1 (tc:max-list (map cdr (vars-in expected)))))))
               (rt (renaming (tc:root attr)))
               (delta-substl ((tc:mgu md) expected rt)))
             (if (null? delta-substl)
                 ;; no result substitution
                 (begin
                   (tc:note-error!)
                   (recurse (cdr hs)
                            (append attrs
                                    (list (tc:node (gen-error-mismatch syn expected rt)
                                                   '())))))
                 (begin
                   (tc:set-current-subst-list!
                    (list
                     (tc:subst-compose
                      (tc:get-current-subst)
                      (car delta-substl))))
                   (let ((t2 ((app (tc:get-current-subst)) rt)))
                     (recurse (cdr hs)
                              (append attrs
                                      (list (tc:node t2 (tc:leaves attr))))))))))))
```

given variable to a value within the global unification environment (lines 1-2). If resolution is successful and the returned binding does indeed contain a type environment, we rewrite the current judgement using the normal form and process it as usual (lines 5-7). A bad binding here is considered a fatal error and type checking terminates (line 8).

<*VariableEnviornmentHypothesisCase* 3.3.4> ≡

```
(let ((te ((app (tc:get-current-subst))
           (to-term lvar))))
  (cond
   ((is-dec-type-expr? te)
    (recurse (cons (tc:hyp (:-d (dec-type-expr->env te) attr-pair))
                   (cdr hs))
             attrs))
   (else (error "tc:sequence:␣Bad␣binding␣for␣logical␣var:␣" lvar))))
```

3.3.5 Subtyping Judgements

The current type helpers support a primitive notion of subtyping. Our restricted subtyping stems from the type system specified for TypedScm00 in [JL96]. It defines two special types, `datum` and `poof`, which are used as the top-most (i.e., most super) and bottom-most (i.e., least specific) types, respectively. All other types exist directly between `datum` and `poof`. Figure 3.12 depicts the hierarchy.

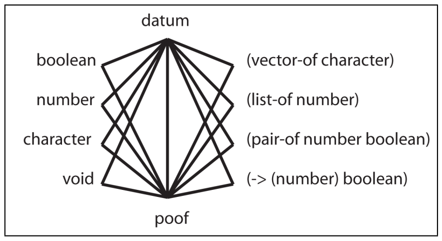


Figure 3.12 TypedScm00 Type Hierarchy [JL96]

Given that the type system to be plugged into the type checking engine for TypedScm05 would also only require a bare minimum in terms of subtyping, support for more advanced subtyping was not included in the initial design of the type helpers.

The actual processing of subtyping cases makes use of an additional procedure shown in <*tc:check–subtype–judgement* 3.3.5>. The simple subtyping case just invokes this procedure with the types it was given and two new logical variables, which are not used further here:

<*SimpleSubtypeCase* 3.3.5> ≡

```
(check-subtype-judgement subtype supertype
                         (tc:new-logicalvar) (tc:new-logicalvar))
```

Judgements making use of the `<:>` constructor are handled specially. Recall that judgements of this form look like `(<:> t1 t2 sub-var super-var)`. The intended behavior is to verify that either `t1` is a subtype of `t2` or vice versa. The implementation for this begins in <*EitherSubtypeCase* 3.3.5> by saving a copy of the current global environment to be restored if needed (line 1) and by testing whether `t1` is a subtype of `t2` (line 2). If this check ends in success, execution continues to the next `tc:mixed` form. However, if the first test fails, we must restore the global environment to its original status and test the opposite relation (lines 7-10).

< *EitherSubtypeCase* 3.3.5 > ≡

```
   (let ((saved-subst-list (tc:get-current-subst-list))
         (t1-sub-result (check-subtype-judgement t1 t2 subtype-var supertype-var)))
     (if (not (tc:error-noted?))
         ;; t1 <: t2 passed, pass along answer
5        t1-sub-result
         ;; t1 <: t2 failed, try other way
         (begin
           ;; restore subst-list
           (tc:set-current-subst-list! saved-subst-list)
10         (check-subtype-judgement t2 t1 subtype-var supertype-var))))
```

We rely on < *tc:check−subtype−judgement* 3.3.5 > for the bulk of the work in subtype processing. In essence it requires the same four parameters as the `<:>` judgement. The first two are used for the actual test, and the second two are logical variables that are to be bound as result-like values so that the supertype and subtypes can be used elsewhere inside rule definitions easily. When checking a subtyping judgement, the two types must first be resolved in the global unification environment (lines 8-10). The unification algorithm for subtyping is then invoked upon the resolved types (line 11). At present, the algorithm implemented by `tc:subtype-mgu` is the same as that described in [JL96], but could feasibly be extended to allow for a more general treatment of subtyping. When a subtype relationship is successful, the unification environment is updated with any constraints gathered as a result of subtype unification and bindings which provide correct values for the two logical variables given as parameters (lines 19-29).

3.4 Summary

The various data structures and type helping procedures described in this chapter comprise an abstract type checking engine that can be used to specify a type checking system for an arbitrary language. The variety of mechanisms made available to programmers provide enough support to implement common type inference rules. Effectively, the type helpers construct type derivation trees based on the rules to accomplish checking[6]. The syntax chosen for interaction with these interfaces is designed explicitly to correspond to a type checker implementer's concept of typing rules.

Most importantly, a separation of concerns within the type checker is accomplished under this design. All code manipulating type judgements and rules exists solely within this module. Users of the engine provide definitions for those procedures required by the method dictionary in external modules. A detailed look at the structure of these external components is the focus of the following chapter.

[6]Appendix B completes the partial derivation used in this chapter.

<tc:check−subtype−judgement 3.3.5*>* ≡

```
(define tc:check-subtype-judgement
  (lambda (hs md app bind attrs recurse)
    (lambda (subtype supertype subtype-var supertype-var)
      (if (tc:error-noted?)
          ;; Type error already, so just pass it on
          <ContinueCase 3.3.2>
          ;; Attempt subtype test, first apply current substitution to vars
          (let* ((cur-app (app (tc:get-current-subst)))
                 (subtype-val (cur-app subtype))
                 (supertype-val (cur-app supertype)))
            (let ((subtype-substl ((tc:subtype-mgu md) subtype-val supertype-val)))
              (if (null? subtype-substl)
                  ;;  subtype relationship failure
                  (begin
                    ;; Update unification environment
                    (tc:note-error!)
                    <ContinueCase 3.3.2> )
                  ;;  subtype relation is valid, pass on subst
                  (begin
                    ;; Update unification environment
                    (tc:set-current-subst-list!
                     (list
                      (tc:subst-compose
                       (tc:get-current-subst)
                       (tc:subst-compose
                        (car subtype-substl)
                        (tc:subst-compose
                         (bind subtype-var subtype-val)
                         (bind supertype-var supertype-val))))))
                    (recurse (cdr hs) attrs))))))))
```

CHAPTER 4. USER DEFINED COMPONENTS

The previous chapters have alluded to the external modules which must be added to the type checking engine to completely specify a type checker. However, most examples have ignored these details (the type checker outlined for the lambda calculus pretended as if they did not exist). Here we more closely examine each of the typical components needed and a portion of their specific implementations in the TypedScm05 system. In essence, this allows users to customize the type checking engine to meet the needs of their system. An example set of type expressions from TypedScm05 appears in section 4.1. Section 4.2 discusses each of the procedures contained within the method dictionary. Sample annotation rules of interest are shown in section 4.3, and an example strategy for error generation in this system is given in section 4.4.

4.1 Type Expressions

Any type checker must provide a representation of data types in order to annotate syntax elements. The method dictionary requires a bare minimum of two basic types for checking: variables and errors; however, most type checkers would contain definitions for many other type expressions. We have already seen an example of a minimal type system in the lambda calculus grammar given in the figures of section 3.1. A more advanced grammar for type expressions is that used in the TypedScm05 implementation, which defines eleven different types. Its datatype definition is shown in Figure 4.1.

```
(define-datatype tc:type-expr tc:type-expr?
  (tc:basic-type-expr (symbol symbol?))
  (tc:function-type-expr
   (arg-types (list-of tc:type-expr?)) (result-type tc:type-expr?))
  (tc:intersection-type-expr (conjoined-types (list-of tc:type-expr?)))
  (tc:type-predicate-for-type-expr (type tc:type-expr?))
  (tc:variant-record-type-expr
   (variants (list-of tc:type-expr?)))
  (tc:applied-type-expr
   (operator-type tc:type-expr?) (operand-types (list-of tc:type-expr?)))
  (tc:variant-type-expr
   (variant-name symbol?)
   (fields (list-of tc:type-expr?)))
  (tc:field-type-binding-type-expr
   (field-name symbol?) (type tc:type-expr?))
  ;; Those required for type helpers
  (tc:variable-type-expr (lvar tc:logicalvar?))
  (tc:declaration-type-expr (pi (tc:environment-of tc:type-expr?)))
  (tc:error-type-expr (error-record tc:error-record?)))
```

Figure 4.1 TypedScm05 Type Expressions

The first eight variants (lines 2-15) are used to implement the types specified in the notation from TypedScm00 [LCD05], while the latter three add functionality needed in order to use the type helpers. Implementation of a method dictionary for these type expressions is straightforward.

4.2 Method Dictionary

Type checker implementers wishing to use our engine need to specify a number of procedures that allow the abstract system to work with the concrete, user-defined type expressions declared externally. This collection of functionality has earlier been referred to as the method dictionary, but its contents have largely been glossed over. This dictionary is defined as a record containing several procedures used by the inference engine. The type helpers extract the various procedures from a record instance, `md`, as shown in the chunk $<$*UnpackMethodDictionary* 4.2$>$ below.

$<$*UnpackMethodDictionary* 4.2$> \equiv$

```
(cases tc:unifiable-md md
       (tc:unifiable-md-dict
        (to-term get-var subterms same-kind nullSubst bind app
                 contravar-subterms covar-subterms invar-subterms
                 subtype-replace is-intersection-type?
                 find-intersection-subtyping find-intersection-supertyping
                 is-bottom? is-top? is-error? gen-error-mismatch gen-error-rule
                 is-dec-type-expr? dec-type-expr->env)
```

This section enumerates each element of the dictionary, providing insight into its purpose. A sample invocation is used in the explanation of items which are procedures. Throughout, the word *term* is used to refer abstractly to a value that is a type expression. The exact definition of a term is specified concretely by the external module.

The following seven elements are the most basic and would be required for even the most rudimentary type checking. They allow for the manipulation of logical variables and simple terms within the type checking engine.

`(to-term lv)` converts the logical variable, `lv`, into a term that represents the same variable. Typically, such a procedure would just incorporate `lv` as a data member of the new term.

`(get-var t)` extracts a logical variable from a term, `t`. Its value is a maybe type[7] that is `(make-something lv)` when `t` represents a logical variable `lv`, and `(make-nothing)` when it does not.

`(same-kind t s)` is true if and only if `t` and `s` are the same "kind" of term, which is to say that they have the same operator. For example, when `t` and `s` are both error types, `(same-kind t s)` returns true.

`nullSubst` is the empty substitution. It is like the identity function in that, for all logical variables `lv`, `(tc:subst-apply nullSubst lv) = (to-term lv)`. Often the `nullSubst` is used as an initial value for the unification environment.

[7]Like in Haskell, an instance of a *maybe type* represents a value that is either an actual value or nothing at all.

`(bind lv t)` represents a substitution that maps a particular logical variable, `lv`, to a specific term, `t`. In all other cases, this substitution behaves like `nullSubst`. In other words, when the resulting substitution is applied to `lv`, the original term `t` is returned.

`(app s)` transforms the substitution `s` into a function that maps terms to terms. This is particularly useful within the type helpers for direct manipulation of terms, rather than logical variables.

`(subterms t)` is a list of all subterms contained in `t`.

The type helpers allow for subtyping, but it requires several additional procedure definitions from the external modules. These eight method dictionary members specify the functionality related to subtyping.

`(contravar-subterms t)` is a list of all subterms in `t` which are to be considered contravariant in the unification of subtypes. Contravariant subterms are treated in the inverse direction from their containing types with respect to subtyping. For example, assume `A <: B`. Then `(contravar-subterms A)` returns a list of terms that should be checked as supertypes (i.e., `:>`) to the terms in `(contravar-subterms B)`.

`(covar-subterms t)` extracts the covariant subterms of `t`. Covariant subterms are checked in the same direction as their containing types for subtyping purposes.

`(invar-subterms t)` provides the subterms that are invariant in `t`. Invariant subterms are those terms which are not to be handled with the subtype unification algorithm. They are instead checked with the same algorithm used for terms in `(subterms t)`.

`(subtype-replace sub super)` returns a pair, like `(sub . super)`, except that any special replacements due to subtyping have been made. This is useful for the treatment of advanced type constructs that need to be de-sugared to more simple types during the tests for subtyping (e.g., variable arity function types).

`(find-intersection-subtyping intersection sought)` attempts to subtype unify its arguments and returns the result substitution of the first type in `intersection` that unifies with `sought`

`(find-intersection-supertyping sought intersection)` attempts to find a substitution that makes `sought` a subtype of each type contained inside `intersection`. If no such substitution exists, it results in the empty list.

`(is-bottom? t)` returns true when `t` is the bottom type within the type hierarchy.

`(is-top? t)` tests whether or not `t` is the top type.

The final six procedures needed allow the type checker to test and manipulate three special terms: error types, intersection types, and declaration types. These particular terms are the only three required to be defined by the method dictionary procedures aside from polymorphic variable types. However, it is possible to provide stubs for these three procedures in the event that the implementer's type system has no need for them. All other types are entirely left to the user's discretion.

`(is-error? t)` is true only when `t` is an error type.

`(is-intersection-type? t)` tests whether or not `t` is an intersection of types. That is, `t` is thought to have more than one allowable type.

`(is-dec-type-expr? t)` is true when `t` is a type expression that represents a declaration type in the system. These types are useful for specifying rules for syntax elements that result in new variable bindings (e.g., formal parameter lists, definition forms, etc).

`(dec-type-expr->env dec-t)` extracts the environment of bindings contained within the declaration type `dec-t`.

`(gen-error-mismatch syn expected inferred)` returns an error term that may contain an error record for syntax element, `syn`, generated from the `expected` and `inferred` types.

`(gen-error-rule syn trees)` creates an error type expression possibly containing an appropriate error record for when the rule corresponding to `syn` fails. This is used for generating more specific error messages than can be achieved with the basic mismatch generator.

In reality, many of these procedures are commonly implemented while specifying the other external aspects, like type expressions. Thus, the method dictionary is fairly simple to produce. A complete example method dictionary for a lambda calculus type checker is included in appendix A.

4.3 Annotation Rules

The type helpers require one procedure (called `tc:annotate` in TypedScm05) that does not exist as part of the method dictionary. Type annotation rule definitions for the abstract syntax appear in this procedure, and it is probably where the most effort is spent when implementing a type checker. The purpose of the annotation code is to delineate the process by which a type is inferred for any given piece of syntax. For example, we say that the call `(annotate pi e)` results in an annotation for the syntactic element `e` in the type environment `pi`.

We have already seen a couple examples of how this annotation takes place in the sample rules given in chapters 1 and 3. Essentially, for each item of syntax, we must define a rule using one of `tc:axiom`, `tc:rule`, `tc:rule-or`, `tc:rule-if`, and `tc:rule-seq`. In order to get a better picture of how these type helpers are actually used, this section uses real code taken from the TypedScm05 checker to illustrate some interesting uses.

4.3.1 Examples of Simple Rule Use

Often, the rules needed are quite small. They tend to have, at most, a handful of hypotheses and do not need side conditions. The simplest cases, those that require no hypotheses at all, need only declare an axiom. Figure 4.2 defines an axiom for self-evaluating elements in Scheme. Self-evaluating items consist of singular numeric constants (e.g., 42), string literals (e.g., “Brodie”), and assorted other simple data. These data can be typed more or less by just asking a question like, “Is this a number?” Here we need only ask what simple type `e` represents (line 3) and assert that `e` has this type with `tc:axiom`.

```
(tc:self-evaluating
 (position datum)
 (let ((t (tc:infer-simple-datum-type datum)))
   (tc:axiom (:- pi (: e t)))))
```

Figure 4.2 Basic Axiom Use

Another common use for rules arises when a piece of syntax contains a list of subterms that need to be included in the list of hypotheses somehow. One way to handle these cases nicely is to use the `map` procedure to quickly create judgements for each member. For example, an `and` expression in Scheme takes one or more tests as arguments:

```
(and (number? 1029) (symbol? 'IBM) (null? '()))
```

The rule in Figure 4.3 shows how one can automatically generate judgements for each test without multiple explicit calls to `:-`.

```
(tc:and-exp
 (position tests)
 (tc:rule (map (lambda (test)
                 (:- pi (: test *tc:boolean*)))
               tests)
          ;;-----------------------------------
          (:- pi (: e *tc:boolean*))))
```

Figure 4.3 Dealing with Lists of Subterms

4.3.2 Example of Complex Rule Composition

While the above rules serve as nice “toy examples” they do not exploit the full range of the type helpers' features. Combining rules together allows for the specification of logically larger rules and is achieved by using `tc:rule-or`. The rules specified as arguments to `tc:rule-or` are executed one at a time until a rule holds. The annotation code in Figure 4.4 depicts how one can nest rules and gives more instances of `tc:rule-if`.

This rule is applied to Scheme lists like: `'()`, `'(2 3 4 5)`, and `'(6 'Hello "World")`. The rule specifies that such lists are empty (lines 9-14), are homogeneous with a type similar to `(list-of number)` (lines 16-29), or that they are heterogeneous and must have the generic type `(list-of datum)` (lines 31-38).

Even complex combinations of rules like one are readily understandable for those reviewing the code. When appropriate comments are provided within the code, the rule notation is quite effective.

4.3.3 Example of Side Definitions with Rule-Seq

For more evidence that the type helpers are useful and convenient, we can turn to a sequential rule definition. Recall that these rules require additional notation allowing for mixing judgements and side procedures in any order. Our specimen of interest for this case is

```
(tc:normal-list-datum
 (position elements)
 (let ((t (tc:new-variable-type-expr))
       (ts (tc:listof-new-variable-type-expr (length elements))))
   (tc:rule-or
    ;; no elements
    (tc:rule-if '()
                (lambda (ts) (null? elements))
                (lambda (ts) (tc:type-expr-null-subst))
                ;; ----------------------------------
                (:- pi (: dat (tc:make-applied-type-expr
                               (tc:make-basic-type-expr 'list-of)
                               (list (tc:new-variable-type-expr))))))
    ;; all elements have the same type, t
    (tc:rule-if (map (lambda (element)
                       (:- pi (: element t)))
                     elements)
                (lambda (ts) (tc:all (lambda (x) (equal? (car ts) x)) (cdr ts)))
                (lambda (ts) (tc:type-expr-null-subst))
                ;; ------------------------------------------
                (:- pi (: dat (tc:make-applied-type-expr
                               (tc:make-basic-type-expr 'list-of)
                               (list t)))))
    ;; elements have different types
    (tc:rule (map (lambda (element type)
                    (:- pi (: element type)))
                  elements ts)
             ;; ------------------------------------------
             (:- pi (: dat (tc:make-applied-type-expr
                            (tc:make-basic-type-expr 'list-of)
                            (list *tc:datum*))))))))
```

Figure 4.4 Combining Rules using tc:rule-or

```
(tc:file
 (position path)
 (let ((mod-sym (tc:extract-name path))
       (prog-exp (tc:parse-scm-file path position))
       (file-type (tc:new-variable-type-expr))
       (mod-env-var (tc:new-logicalvar)))
   (tc:rule-seq (list
                 (tc:hyp (:- (tc:env-empty) (: prog-exp file-type)))
                 (tc:def
                  (lambda (ts)
                    (let* ((file-env (tc:declaration-type-expr->env (car ts)))
                           (mod-env-type-expr (tc:binding->type-expr
                                               (tc:env-value mod-sym file-env))))
                      (tc:type-expr-bind mod-env-var
                                         mod-env-type-expr)))))
                ;;------------------------------------------------------------
                (:- pi (: modname (tc:variable-type-expr mod-env-var))))))
```

Figure 4.5 Using Side Definitions to Add Bindings

a syntactic addition made to Scheme that allows for importing modules (more on the specifics of modules appears in chapter 5) shown in Figure 4.5.

The concrete syntax here is not as important as the semantic meaning associated with `tc:file`. Upon seeing one of these elements, we need to first check the contents of the file for which it is named without using the type environment `pi` (line 8). Then we must extract the definitions given for this module (lines 9-10) and provide them in the resulting type (line 12-13). A `tc:def` is used here in order to grant access to the loaded module's environment in the conclusion judgement through the logical variable `mod-env-var`.

All told, over 100 rule and axiom definitions make up the annotation procedures used for TypedScm05. Despite its size, managing the code is a relatively simple process due to the rule-based strategy. For any given element of syntax, one need only alter its corresponding rule. Adding new syntax is a similarly easy process: after adding the necessary abstract syntax a new rule is added to the annotation code. Anecdotally speaking, the `has-type` expression described in [LCD05] was added to TypedScm05 by someone not familiar with the bulk of the rule implementation, and the whole process required was done in approximately an hour.

4.4 Error Handling

One of the most important parts of any type checker is its ability to generate meaningful error messages once a discrepancy is detected. The type checking engine permits a great deal of flexibility in terms of errors. Programmers of the external modules may specify highly advanced error generation routines, or ones that do very little. In fact, it is possible to define one generic error type generator that is used for all errors and nothing more, though in practice, this would not be very useful. TypedScm05 provides a good example of how such processing could proceed.

4.4.1 Error Generation

TypedScm05's error creation process is somewhat similar to type annotation. Given any particular piece of syntax, a list of the annotation results associated with its subexpressions, and knowledge of the underlying inference rule, we determine what situation caused the error to occur. We then encapsulate the necessary information into an error record. These error records are another variant type where each form corresponds to particular message to be displayed for the end user. They store pertinent data so that a proper error message can be generated from them following the completion of type checking.

Consider the annotation rule example given for conditional expressions in Figure 1.4. There exist three possible scenarios that can result from execution of this rule. First, it is possible that a non-boolean value was specified for the test subexpression. Second, it may be the case that some general type error was found in the test or consequent and alternate arms. Lastly, it is feasible that no subexpression had an error, but that the consequent and alternate types were found incompatible during checking of the subtype relationship. Any one of these cases will produce an error in the type checking engine and will result in a call to the implementer's gen-error-rule procedure.

The code in Figure 4.6 illustrates how it is possible to dissect the information given by the engine and contained in the annotated syntax trees to create the appropriate error record for a conditional expression. Lines 8-9 handle the situation where a bad test expression is provided, while lines 10 and 11 deal with other generic errors that might arise in the subexpressions.

If checking otherwise failed, it must have been caused by a bad subtype relationship and is handled in lines 12-15. For each of these different errors we define a special error record (e.g., `tc:if-subtype-error-record`) that can be used to produce a meaningful error message at a later point. A composite error record form is allowed to concatenate multiple errors together.

```
   (tc:conditional-exp
    (position test consequent alternate)
    (let ((test-type (car tree-tops)))
      (cond
5      ((and (tc:error-type-expr? test-type)
             (tc:mismatch-error-record?
              (tc:error-type-expr->error-record test-type)))
        (tc:error-type-expr (tc:badtest-error-record
                             (tc:error-type-expr->error-record test-type))))
10     ((tc:contains-errors? tree-tops)
        (tc:composite-error-maker tree-tops))
       (else
        (tc:error-type-expr
         (tc:if-subtype-error-record consequent (cadr tree-tops)
15                                   alternate (caddr tree-tops)))))))
```

Figure 4.6 Example Error Generation for Conditional

The process of generating error records for each of the special cases in TypedScm05 follows the basic pattern illustrated in the above code. All errors for which no special message is generated are handled by a generic routine that propagates the messages onward.

4.4.2 Error Output

Completely separate from the type checking engine is the process by which errors are presented to the type checker's end user. In the event of an error, the type checking engine will produce an abstract error record like that described in the previous section. There are any number of ways in which this information can be conveyed to a user. The current implementation of TypedScm05 uses it to produce a textual error message nearly identical to the messages given in TypedScm00. A sample error message for an instance of `tc:if-subtype-error-record` (i.e., the case when a conditional expression's subtyping judgement is violated) appears in Figure 4.7.

```
typed> (if (number? 1919) 'AEA "Bohumil")
<standard input>: line 2: Arms of if expression have different types
  line 2: Left  arm: (quote aea)
  line 2: Right arm: "Bohumil"
  Left  arm's type: symbol
  Right arm's type: string
```

Figure 4.7 Example Error: Bad Subtying Relationship on Conditional

The output mechanism is trivial since all that need be done is to add necessary textual information to what is already contained in the error records corresponding to a particular type of error. Additional future possibilities for error output are discussed later in Chapter 7.

CHAPTER 5. LANGUAGE AND TYPING EXTENSIONS

Beyond the benefits which stem from an implementation that separates the various concerns within the type checker, TypedScm05 improves upon the previous type system. This chapter explores these enrichments in detail. Section 5.1 delves into the largest addition, a language-level module system. Variations on inference rules and the value they add are investigated in section 5.2. The chapter ends with section 5.3 and a discussion of corrections made to variant record typing. Aside from the extensions mentioned here, the type system is the same as in TypedScm00; the reader is directed to [JL96] for formal specifications.

5.1 Module System

The language designated in TypedScm05 adds modules to that specified in TypedScm00. "A module defines a scope for names (and syntax), and hides all names (and syntax) that it does not export explicitly [LCD05, pg. 19]." There are many benefits to using a module system, such as the separation of interfaces and implementation, independent compilation, and enhanced code reusability [Que03]. However, the Scheme standard does not include a definition for modules.

We extend the language to allow for a simple module system that is typeable. Our module syntax is a restricted version of that given in MzScheme [Fla04]. It permits a single `provide` form and limits the user to at most one `require` form. Formally, the syntax is defined by the grammar in Figure 5.1 [LCD05].

Other than the use of `deftype` and `defrep`, the semantic meaning for a module remains that of MzScheme. The notion of a `defrep` is identical to that used in TypedScm00 and allows for the declaration an internal representation for an abstract datatype; however, such declarations may now only appear inside of a module. This requirement, combined with the ability to restrict exports using a `provide` clause, allows for a greater degree of clarity when implementing ADTs.

An example module definition for a `box` abstract data type whose internal representation is a `(vector-of number)` appears in Figure 5.2.

Typing a module definition requires the use of `tc:rule-seq` and a large number of mixed hypotheses. In lieu of listing the entire code here, we will give a general overview to the approach taken. Steps resulting in the creation of a type environment define a name to be used to refer to it in subsequent stages.

1. Import all definitions given inside `require` forms and combine these types into an environment, e_1

2. Retrieve the abstract types provided by the `deftype` forms and save them for later in an environment, e_2

⟨module⟩ ::= (`module` ⟨identifier⟩
  | ⟨initial-required-module-name⟩
  | ⟨provide⟩
  | {⟨deftype⟩}*
  | ⟨require-for-syntax-opt⟩
  | ⟨require-opt⟩
  | ⟨defrep-opt⟩
  | {⟨definition⟩}*)
⟨initial-required-module-name⟩ ::= ⟨module-name⟩
⟨module-name⟩ ::= ⟨identifier⟩
  | ⟨unix-relative-path-string⟩
  | (`file` ⟨path-string⟩)
  | (`lib` ⟨filename-string⟩ {⟨collection-string⟩}*)
⟨unix-relative-path-string⟩ ::= ⟨string⟩
⟨path-string⟩ ::= ⟨string⟩
⟨filename-string⟩ ::= ⟨string⟩
⟨collection-string⟩ ::= ⟨string⟩
⟨defrep-opt⟩ ::= | ⟨defrep⟩
⟨require-for-syntax-opt⟩ ::= | ⟨require-for-syntax⟩
⟨require-for-syntax⟩ ::= (`require-for-syntax` ⟨require-spec⟩)
⟨require-opt⟩ ::= | ⟨require⟩
⟨require⟩ ::= (`require` {⟨require-spec⟩}+)
⟨require-spec⟩ ::= ⟨module-name⟩
⟨provide⟩ ::= (`provide` {⟨identifier⟩}*)

Figure 5.1 TypedScm05 Module Syntax

```
(module module-d (lib "typedscm.ss" "lib342")
  (provide make-box box->value box-set!)
  (deftype make-box (-> (number) box))
  (deftype box->value (-> (box) number))
  (deftype box-set! (-> (box number) void))
  (require (lib "module-a.scm" "lib342"))
  (defrep (box (vector-of number)))
  (define make-box (lambda (n) (vector n)))
  (define box->value (lambda (b) (vector-ref b 0)))
  (define box-set! (lambda (b v) (vector-set! b 0 v))))
```

Figure 5.2 Module for Box ADT

3. Translate the abstract types given in the `deftypes` into their corresponding concrete types based on the `defrep` information and store the bindings in an environment, e_3
4. Merge environments e_1 and e_3 to form the environment e_4 to be used for inference on the definition forms
5. Infer the types of the internal definitions using e_4
6. Merge the bindings gleaned in step 5 with those in e_1 and e_2 such that bindings from e_2 have highest precedence, resulting in e_5
7. Filter out all bindings from e_5 which do not appear in the `provide` list to form e_6
8. Return a declaration type containing the a binding from the module name to the environment e_6

When the bindings contained with in a module are needed later for a `require` form, the type checker simply performs a lookup of the module name and unpacks values found in the corresponding environment.

5.2 Bidirectional Type Checking

The type notation defined for both TypedScm00 and TypedScm05 allows programmers to assign a particular type to an identifier using a `deftype` definition. The syntax for this top level form is given below [LCD05].

⟨deftype⟩ ::= (**`deftype`** ⟨name⟩ ⟨type-exp⟩)
⟨name⟩ ::= ⟨identifier⟩

The intent is for the type checker to generate an error message if the inferred type of a subsequent definition of the same name does not match the given type. However, in practice the TypedScm00 checker does not make full use of the information given during type checking. At best, it can only issue a message saying that the given and inferred types do not match for a procedure. For example, consider the definitions:

```
(deftype f (-> (boolean number) number))
(define f (lambda (t x)
            (if x t 9)))
```

This code contains an intentional error on line 3; the first and second arguments to the if-expression have been mistakenly swapped. The error produced by TypedScm00 is shown in Figure 5.3. While it does detect that an error occurred, the message generated does little to point the programmer to the position of the real error.

This less than ideal error message arises from the fact that standard inference rules for procedure definitions do not take into account the constraints placed on arguments by a previous type definition. Type inference occurs in a bottom-up fashion and the comparison is

```
<standard input>: line 5: Type mismatch between inferred and expected types
  Syntax: (lambda (t x) (if x t 9))
  Expected: (-> (boolean number) number)
  Inferred: (-> (number boolean) number)
```

Figure 5.3 Error Message in TypedScm00

only made when a top-level type has been determined. However, in the case where a previous deftype exists, it is reasonable to assume the `deftype` is correct and use it while processing the body of the definition. Such an approach would allow the type checker to point out that `x` and `t` are, in fact, incorrectly used in the above definition (see Figure 5.4).

```
<standard input>: line 4: Wrong type for test expression
  Test expression: x
  Expected: boolean
  Inferred: number
```

Figure 5.4 Error Message in TypedScm05

To achieve this behavior in TypedScm05 we make use of a bidirectional checking scheme. Pierce and Turner introduced this strategy as a refinement on the basic typing inference algorithm [PT98a]. They provide for two distinct modes within the type checker: "synthesis mode, where typing information is propagated upward from subexpressions, and checking mode, where information is propagated downward from enclosing expressions [PT98a, pg. 16]." In their system, synthesis mode corresponds to the standard typing rules and is used when no previous information is present. Thus, if given no type definitions, processing proceeds using basic inference. Checking mode, on the other hand, is applied when the context determines the type of the expression. In such cases, the system simply needs to verify that the expression does indeed have the correct type.

If checking mode were applied to our above code example, a more useful error message would be expected. This is possible because the restrictions that $t\!:\!boolean$ and $x\!:\!number$ can be propagated down into the processing of the lambda body. The annotation rules implemented in TypedScm05 leverage bidirectional checking where possible for this reason. Procedure definitions like these are encapsulated into special named-lambda expressions and handled with the following synthesis and checking rules:

$$\textsc{Syn-Named-Lambda}\quad \frac{\Pi,\ x_1 : \sigma_1,\ \ldots,\ x_n : \sigma_n \vdash e : \tau}{\Pi \vdash (\mathit{nlambda\ name}\ (x_1 \ldots x_n)\ e) : (\rightarrow (\sigma_1 \ldots \sigma_n)\ \tau)}$$

$$\textsc{Chk-Named-Lambda}\quad \frac{\Pi,\ \mathit{name} : (\rightarrow (\sigma_1 \ldots \sigma_n)\ \tau),\ x_1 : \sigma_1,\ \ldots,\ x_n : \sigma_n \vdash e : \tau}{\Pi,\ \mathit{name} : (\rightarrow (\sigma_1 \ldots \sigma_n)\ \tau) \vdash (\mathit{nlambda\ name}\ (x_1 \ldots x_n)\ e) : (\rightarrow (\sigma_1 \ldots \sigma_n)\ \tau)}$$

In the checking rule, the system extracts the argument type information already known and places it into the environment for checking the body expression. As expected this tactic produces a more precise error message that can be used to locate errors. Figure 5.4 shows the message generated for the same two definitions. Notice that this message points the programmer to the specific expression suspected of being the source of the problem (the misuse of argument x).

One consideration to make with top-down propagation is that incorrect type declarations could report false errors. However, this is not an unusual behavior as it is commonplace among statically-typed languages with explicit type declarations. It is, as always, up to the programmer to interpret error messages. Nonetheless, this strategy tends to more accurately direct novice programmers to the source of their common mistakes.

5.3 Variant Record Typing

TypedScm00 added syntax to Scheme that provides for variant record types. This syntax was adapted from [FWH01] and incorporates both the `define-datatype` definition form and the `cases` expression. The old system did not perform type checking on these record types and thus limited its usability in more advanced educational settings. TypedScm05 makes no new alterations to the syntactic forms specifically, but it does allow for proper verification of these advanced types. Checking a `define-datatype` declaration results in bindings for the variant record type and the types of the variant constructors. Consider the example definition from [LCD05]:

```
(define-datatype person person?
   (student (name string?) (major string?))
   (professor (name string?) (office number?)))
```

This has the effect of the following several `deftype` declarations:

```
(deftype person? (type-predicate-for person))
(deftype student (-> (string string) person))
(deftype professor (-> (string number) person))
(deftype person
   (variant-record
      (student (name string) (major string))
      (professor (name string) (office number))))
```

Variants of type `person` can be constructed and used in a typeable `cases` expression:

```
(professor "Gary␣Leavens" 229) : person

(cases person (student "Brian␣Dorn" "Computer␣Science")
  (student (name major) name)
  (professor (name office) (string-append "Dr.␣" name))) : string
```

The bidirectional approach presented in the previous section is a key aspect of the means by which `cases` expressions are typed. Bottom-up type synthesis for variant record types is a non-trivial problem in general as noted in [Wan87], [Rém89], and others. However, we can reframe the specific problem here to use a top-down checking strategy. Just as is the case when a procedure's type is known from a previous `deftype`, variant types are known due to previous `define-datatype` definitions. Using the types defined for the variant record and the various constructors, top-down verification of the clauses is a simple task.

CHAPTER 6. IMPLEMENTATION CHALLENGES

Throughout the implementation of TypedScm05, a number of significant problems were encountered. Not all of these obstacles warrant mention here; however, three issues resulted in considerable design changes. A brief discussion of each and their impact on the final design is provided in this chapter. Considerations between lazy and eager evaluation are summarized in section 6.1. Complications with the unification environment appear in section 6.2, and section 6.3 reflects upon runtime performance issues.

6.1 Delayed Evaluation

Chapter 3 introduces the type helper system as being based upon a prototype written in the Haskell programming language. Like Scheme, Haskell is a functional language, but it incorporates a lazy evaluation mechanism. *Lazy evaluation* refers to a just-in-time strategy for performing computations. In such systems, arguments to procedures are frozen until the results are explicitly needed. This is in contrast to *eager evaluation,* like that employed in Scheme. Here, all argument values are computed immediately, regardless of whether or not they are actually used at some later time.

The different evaluation methods were not taken into consideration during the initial code translation from Haskell to Scheme. Not surprisingly, the resulting program did not function as anticipated. The root problem here was that not all rules and judgements should be computed immediately when they appear in the annotation code. This situation arises primarily in the case of `tc:rule-or` and `tc:rule-seq`. These two helpers require, by definition, that their arguments are processed in a sequential fashion, and thus eager evaluation causes errors. As an example, consider a rule for variable references:

```scheme
(tc:le-varref
 (variable)
 (rule-if
  '()            ;; no hypotheses here
  (lambda (ts)
    (tc:env-bound? variable pi))
  (lambda (ts) (tc:type-expr-null-subst))
  (:- pi (: e (tc:env-value variable pi)))))
```

Here the evaluation of the expected type in the conclusion should only proceed when the side condition holds true. If the variable is not bound in the environment, an attempt to procure its value will cause a run-time error. Granted, it is possible to rewrite this rule using the optional side definition procedure so as to not perform the lookup in the conclusion. However, such tricks are not always possible in general and thus delayed evaluation is required.

In order to achieve proper function, it was necessary to add explicit delay expressions around arguments to these procedures, as well as the judgement constructor `:-`. The Scheme macros shown below were used in order to preserve the desired notation and alleviate any programmer burden caused by managing evaluation concerns.

```
(define-syntax tc:rule-seq
  (syntax-rules (list)
    ((tc:rule-seq (list h1 ...) conc)
     (tc:rule-seq-local (list (delay h1) ...) conc))))

(define-syntax tc:rule-or
  (syntax-rules ()
    ((tc:rule-or t1 ...)
     (tc:rule-or-local (delay t1) ...))))

(define-syntax :-
  (syntax-rules (:)
    ((:- pi (: s t))
     (:-d pi (: s (delay t))))))
```

In conjunction with proper calls to force subsequent evaluation, these simple syntax macros accommodate our need for some degree of delayed evaluation within the system.

6.2 Unification Environment

Unfortunately, not all errors could be addressed as mere differences between the two languages involved in the translation. A number of more severe deficiencies were discovered in the original prototype when the type checking engine was applied to non-trivial examples. Of these, the most significant was the manner in which the unification environment was managed.

The original design called for the unification environment, otherwise referred to internally as the current substitution, to be threaded through calls to the type helpers and the annotation procedure. At some point, the current substitution must be initialized, and this took place within `tc:rule-seq-helper` upon each call to the procedure `tc:rule-sequence`. The problem with this approach only became apparent when large syntax trees were annotated. When examples grew large (i.e., they contained many levels within their syntax trees), typing anomalies were detected. The fundamental flaw here was that constraints gathered from rule processing across a particular level of the tree were lost because each invocation of the type helpers on subexpressions was treated semi-independently.

Creation of a single global substitution list and elimination of the threading not only preserved all constraints correctly, but it also greatly simplified the code within the type checking engine. An external interface is now exported from the type helper module so that the global environment can be initialized once and only once upon the first call to the annotation routine.

6.3 Runtime Speed

Of the major problems in TypedScm05, slow execution times proved the most frustrating. The data structures originally specified in the Haskell prototype were too simple to support reasonably efficient execution. This version designed substitutions as finite functions from logical variables to terms and represented the variable resolution process as an application of these functions. These substitutions are the building blocks for the environment used during type checking. Creating a unification environment with this style equates to the composition of all individual functions into one large function. While this stores sufficient information needed for the resolution of bindings, the application process can be time consuming. Consider a hypothetical composition of six functions:

$$ue(x) = f(g(h(i(j(k(x))))))$$

where the function that resolves a specific logical variable y is $f(x)$. In this situation the application of the unification environment to y, $ue(y)$, requires that all six functions be traversed (from innermost to outermost) prior to finding an answer. Likewise, unbound variable resolution requires computation of all functions before one can determine that no binding exists in the environment. Essentially application of a substitution in this scheme boils down to a linear search, whose average and worst case run-times are $O(n)$.

For small examples, a linear run-time penalty is not noticeable to the user; however, code to be typechecked need not grow too large before this becomes a very real factor for usability. This is due to the fact that a large number of logical variables (and bindings) are used in the creation and processing of the annotation rules.

To overcome this fundamental speed limitation, the functional representation was replaced with one that uses hash tables to store and retrieve bindings. A major design concern here was the need to preserve the same notions of substitutions and their application. An interface was designed to achieve this goal as well as the need for the abstraction of substitutions. It allows for the creation of a null substitution, instantiation of a substitution for a specific binding, application of a substitution to a variable, and substitution composition.

Internally, substitutions are represented by a variant record type that maintains a hash table containing the relevant bindings. In this sense, the null substitution encapsulates an empty table; application is emulated by a value lookup in the hash table; and composition requires the merge of two tables. While the composition task here is less efficient than that used in the functional representation, most of the time spent on substitutions is in application. This is where the hash table version shines. Its $O(1)$ average case run-time produces marked speedup for almost any example.

While speed remains a minor issue in the current version of TypedScm05, the system now runs at a tolerable speed for reasonably large Scheme programs. The current system's point of slowdown is in the computation of a fixed point when logical variables are resolved in the unification environment. Remember that these logical variables may, and often times do, refer to other logical variables. What can result is a potentially large chain of variables that must be sequentially processed until a final result is reached. We believe this can be addressed in the future by standard techniques, such as incorporating a union-find algorithm to short-circuit variable resolution.

CHAPTER 7. FUTURE WORK

TypedScm05 provides a number of possibilities for future work. The current implementation supports the DrScheme/MzScheme and the ChezScheme interpreters, but it is possible that it could be further extended to work with other systems (SCM, etc.). Within the currently supported programming environments, there is an opportunity to more tightly incorporate the type checker in order to leverage specific environmental features. There is also need to conduct experimental evaluations to determine whether or not a static type checker actually does make learning Scheme easier for novices.

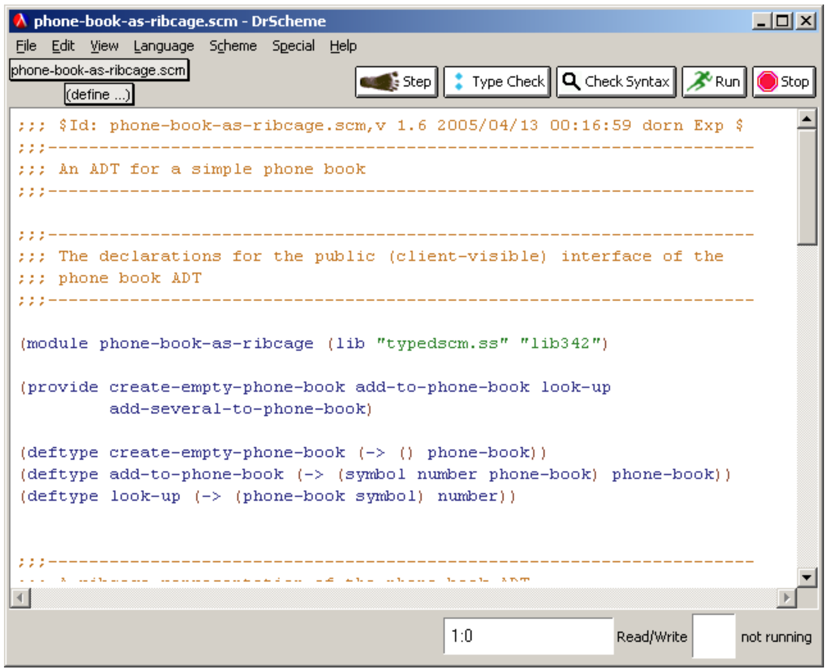
7.1 DrScheme Integration

Up to this point, we have largely ignored the programming environments in which the type checker will running. However, special considerations were made during implementation that allow for future extension, especially with one particular system. DrScheme is "a comprehensive programming environment for Scheme [that] fully integrates a graphics-enriched editor, a multilingual parser that can process a hierarchy of syntactically restrictive variants of Scheme, a functional read-eval-print loop, and an algebraically sensible printer [FFF^{+}97, pg. 369]." It was design specifically as a pedagogic sand box for beginning Scheme programmers.

Among DrScheme's features is the ability to extend the programming environment itself using a tool mechanism. One such tool is the syntax checker that comes standard with DrScheme. This tool allows on-demand processing of a program's lexical structure. It can highlight errors in a student's program and provide other useful syntax annotations. Invoking the syntax checker is as simple as clicking the appropriate toolbar button. Figure 7.1 shows a screen capture of the interface.

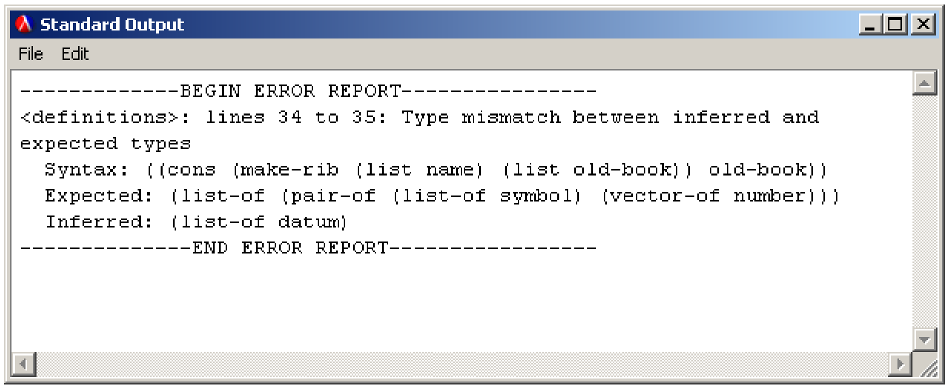
Ideally we would like to provide functionality similar to the syntax checker using TypedScm05. As of the time of writing, DrScheme integration of the type checker is minimally supported. The current tool incorporates a language definition for our extensions and adds a button for type checking on the toolbar (see Figure 7.1). Clicking the "Type Check" button invokes the type checker and processes the results. If the program is type correct, the user gets no error information. Type errors that are detected are displayed in the same textual format is used when the type checker is in interactive mode, but they appear in a separate window as seen in Figure 7.2.

Though this tool is already useful for students using the DrScheme environment, there are many possibilities for additional features. For example, it would aid students in error location if the type checker tool could highlight expressions containing errors much like the syntax checker does. Fortunately, the error record generation and output mechanisms discussed in Chapter 4 make this matter of simply providing the DrScheme library functions with the appropriate information.



```
;;; $Id: phone-book-as-ribcage.scm,v 1.6 2005/04/13 00:16:59 dorn Exp $
;;;---------------------------------------------------------------------
;;; An ADT for a simple phone book
;;;---------------------------------------------------------------------

;;;---------------------------------------------------------------------
;;; The declarations for the public (client-visible) interface of the
;;; phone book ADT
;;;---------------------------------------------------------------------

(module phone-book-as-ribcage (lib "typedscm.ss" "lib342")

(provide create-empty-phone-book add-to-phone-book look-up
         add-several-to-phone-book)

(deftype create-empty-phone-book (-> () phone-book))
(deftype add-to-phone-book (-> (symbol number phone-book) phone-book))
(deftype look-up (-> (phone-book symbol) number))


;;;---------------------------------------------------------------------
```

Figure 7.1 DrScheme and the TypedScm Tool



```
-------------BEGIN ERROR REPORT----------------
<definitions>: lines 34 to 35: Type mismatch between inferred and
expected types
  Syntax: ((cons (make-rib (list name) (list old-book)) old-book))
  Expected: (list-of (pair-of (list-of symbol) (vector-of number)))
  Inferred: (list-of datum)
--------------END ERROR REPORT-----------------
```

Figure 7.2 DrScheme Type Error Output

7.2 Experimental Study

It is important that we evaluate the educational effectiveness of our type checker given that its original design was to ease the learning process for students. Specifically, it would be pertinent to explore how static typing affects the number of latent errors in student code and how interaction with the type checker impacts retention of typing concepts.

We believe that many of the common errors made by novices result from incomplete mental models of the system being programmed and the language being used, especially since these issues have been seen to decrease in frequency with experience. When a student is able to write code that produces the correct output but has latent type errors (a common situation with new students using dynamically-typed languages), their incorrect or incomplete mental model persists. A static type checker disallows such errors and confronts students with any potential misunderstandings they may have. Developing an ability to recognize and overcome misconceptions by oneself is an integral part of the learning process [WM98]. Thus, it is our belief that the use of a statically-typed language for introductory programming experiences more quickly develops an accurate intuition about a program's correctness and makes a student more aware of typing in future applications, even when static type checking may not be available.

Previous research [Gan77] [PT98b] has held that the use of a static type checker increases programmer productivity and reduces the number of latent errors in finished products. Gannon [Gan77] also noted that higher performing individuals (as measured by course grades) in his study showed a smaller benefit from type checking than lower performers. This observation supports our suspicion regarding the development of mental models when coupled with the assumption that higher performers typically have a more acute sense of typing going into the experiment.

However, these studies have not examined the after-effects that use of a static type checker has on learners. In fact, they tend to utilize sample groups consisting of relatively experienced programmers (e.g., Ph.D. students). How these systems impact true novices remains a largely unexplored question. Such a study was intended to be incorporated into this paper but was later eliminated due to time constraints in the academic term. It would be a relatively simple matter to design and conduct a pilot study with a future group of ComS 342 students at Iowa State University.

CHAPTER 8. CONCLUSION

We conclude by considering the space in which TypedScm05 resides relative to other systems and recap the overall contributions of this work.

8.1 Related Work

The addition of static type checkers to Scheme and other dynamically-typed languages is certainly not a new phenomenon. There are several other existing systems that address this issue including, but not limited to, SoftScheme [WC93], STYLE [Lin93], SPS [Wan89], MrSpidey [FF99], and of course TypedScm00 [JL96] [LC05]. Each of these different systems has its own unique set of advantages and disadvantages.

For the specific audience with which we are concerned (novice programming students), Jenkins and Leavens provide a convincing argument for the type notation used in TypedScm00 over SoftScheme, Style, and SPS [JL96]. They note that, while the other systems are feature rich and complete, in many cases they would be unusable by typical introductory students due to complex type notation, runtime overhead, and occasional incompatibility with preexisting code. Because it is based on the same fundamental type notation and inference system, TypedScm05 inherits these characteristics.

It differs from TypedScm00 primarily in its implementation approach. While TypedScm00's code is difficult to understand and maintain, our system utilizes an external type checking engine, providing for a greater separation of concerns and enhanced readability. In addition, TypedScm05 produces more usable error messages through use of bidirection, is more complete in its treatment of variant record data types, and introduces a typeable module system.

A lesser goal of TypedScm05 (and an area of future work) is to provide seamless integration with the DrScheme programming environment. Another static analysis tool, MrSpidey [FF99], already provides a great deal of support for DrScheme. This tool, developed directly by the DrScheme team, provides for the declaration of types and for the addition of type assertions within Scheme programs. Unfortunately, it only supports version 103p1 of DrScheme (which has been outdated for quite some time). TypedScm05 functions properly with current versions of DrScheme, and its type checking infrastructure should be compatible with any changes to DrScheme made in the forseeable future.

Though it is easy to find examples of other type checking systems, it is more difficult to find details about the manner in which they are implemented. Our reusable unification engine provides the ability to quickly create type checkers for languages other than Scheme. Another system, TyS, developed at Universidad de Oviedo also addresses the issue of type checker construction [Rod03], but from an object-oriented standpoint. It aims to automatically build object-oriented type checkers that can be incorporated into language processing systems. Like our system, TyS requires programmers to specify a set of rules that are to be used during static analysis. However, rules in this system are given as Java classes rather than in a syntax closely aligned to mathematical rule specification like that of TypedScm05.

8.2 Contributions

This thesis has outlined a number of the features unique to our approach to static type checking in Scheme. We have discussed the specifics of the TypedScm05's design and implementation. Additionally, we have explored the practical benefits it has over previous systems and have demonstrated its extensibility for support in additional programming environments.

We solve the problem of code entanglement by creating an abstract type checking engine comprised of a reusable set of "type helping" operations. These type helpers:

- achieve a modular separation between type checking and unification algorithms, type inference rules, and error message generation,
- export an interface that allows programmers to implement type rules in a manner that closely mirrors the formal way they specify such rules, and
- support a high level of maintainability and readability in type checking code.

The type checker we implemented using this engine is evidence for its usefulness and provides a number of other contributions. Specifically it:

- adds a module system to the typeable language of TypedScm00,
- incorporates bidirectional type checking rules to produce more detailed error messages for declared identifiers,
- offers a semantically correct typing of variant record types, and
- allows for easy future extension of the DrScheme tool for a more integrated approach to displaying type errors.

We look forward to seeing students make use of TypedScm05 and hope that it proves to be as valuable a learning tool for them as its implementation was for us.

APPENDIX A. LAMBDA CALCULUS TYPE CHECKER

```scheme
;; This file provides customizations for a lambda calculus-like type checker
;; with no subtyping.

(require (lib "tc-util.scm" "lib342")
         (lib "maybe.scm" "lib342")
         (lib "tc-subst.scm" "lib342")
         (lib "tc-type-helpers.scm" "lib342")
         (lib "tc-environments.scm" "lib342"))

;; Define our language grammar
(define-datatype tc:lambda-calc tc:lambda-calc?
  (tc:le-varref
   (variable symbol?))
  (tc:le-self-evaluating
   (datum datum?))
  (tc:le-procedure-call
   (operator tc:lambda-calc?) (operand tc:lambda-calc?))
  (tc:le-lambda-exp
   (formal symbol?) (body tc:lambda-calc?)))

;; Define our types and helping procedures
(define-datatype tc:type-expr tc:type-expr?
  (tc:basic-type-expr (symbol symbol?))
  (tc:function-type-expr
   (arg-type tc:type-expr?) (result-type tc:type-expr?))
  ;; For type helpers
  (tc:variable-type-expr (lvar tc:logicalvar?))
  (tc:error-type-expr))

(define tc:new-variable-type-expr
  (lambda ()
    (tc:variable-type-expr (tc:new-logicalvar))))

;; Define our method dictionary
(define tc:null-subst
  (lambda () (tc:make-null-subst tc:variable-type-expr)))

(define tc:type-expr-bind
  (lambda (v t)
      (tc:make-subst v t tc:variable-type-expr)))

(define tc:type-expr-as-unifiable
  (lambda ()
    (tc:unifiable-md-dict
     ;; to-term
     (lambda (v) (tc:variable-type-expr v))
     ;; get-var
     (lambda (te)
       (cases tc:type-expr te
         (tc:variable-type-expr (v) (make-something v))
         (else (make-nothing))))
     ;; subterms
     (lambda (t)
       (cases tc:type-expr t
          (tc:function-type-expr
           (arg-type result-type)
           (cons arg-type (list result-type)))
          (else '()))))
 ;; same-kind
 (lambda (t s)
   (let ((get-kind
          (lambda (te)
            (cases tc:type-expr te
                   (tc:basic-type-expr (symbol) symbol)
                   (tc:function-type-expr (arg-type result-type) 'function)
                   (tc:variable-type-expr (lvar) 'variable)
                   (tc:error-type-expr () 'error)))))
     (eq? (get-kind t) (get-kind s))))
 ;; nullSubst
 (tc:null-subst)
 ;; bind
 tc:type-expr-bind
 ;; app
 (lambda (s)
   (lambda (attrib)
     (letrec ((app-once
                (lambda (t)
                  (cases tc:type-expr t
                         (tc:variable-type-expr
                          (lvar)
                          (tc:subst-apply s lvar))
                         (tc:function-type-expr
                          (arg-type result-type)
                          (tc:function-type-expr (app-once arg-type)
                                                 (app-once result-type)))
                         (else t))))
               (app
                (lambda (attrib applied)
                  (if (equal? attrib applied)
                      applied
                      (app applied (app-once applied))))))
       (app attrib (app-once attrib)))))
 ;; contravar-subterms
 (lambda (t) '())
 ;; covar-subterms
 (lambda (t) '())
 ;; invar-subterms
 (lambda (t) '())
 ;; subtype-replace
 (lambda (sub super) (cons sub super))
 ;; intersection-type?
 (lambda (t) #f)
 ;; find-intersection-subtyping
 (lambda (intersection sought) (tc:null-subst))
 ;; find-intersection-supertyping
 (lambda (sought intersection) (tc:null-subst))
 ;; is-bottom?
 (lambda (t) #f)
 ;; is-top?
 (lambda (t) #f)
 ;; is-error?
 (lambda (t)
   (cases tc:type-expr t
          (tc:error-type-expr () #t)
          (else #f)))
 ;; gen-error-mismatch
 (lambda (syn expected inferred) (tc:error-type-expr))
 ;; gen-error-rule
 (lambda (syn trees) (tc:error-type-expr))
 ;; is-dec-type-expr?
```

```
      (lambda (t) #f)
      ;; dec-type-expr->env
      (lambda (t) #f)
      )))

;; Instantiate the type helpers
(define tc:axiom (tc:axiom-helper (tc:type-expr-as-unifiable)))
(define tc:rule (tc:rule-helper
                 (tc:type-expr-as-unifiable)
                 (lambda (pi e) (tc:lambda-calc-annotate-lower pi e))))
(define tc:rule-if (tc:rule-if-helper
                    (tc:type-expr-as-unifiable)
                    (lambda (pi e) (tc:lambda-calc-annotate-lower pi e))))
(define rule-seq-helper (tc:rule-seq-helper
                         (tc:type-expr-as-unifiable)
                         (lambda (pi e) (tc:lambda-calc-annotate-lower pi e))))
(define rule-or-helper (tc:rule-or-helper
                        (tc:type-expr-as-unifiable)
                        (lambda (pi e) (tc:lambda-calc-annotate-lower pi e))))
(define-syntax tc:rule-seq
  (syntax-rules (list)
    ((tc:rule-seq (list h1 ...) conc)
     (rule-seq-helper (list (delay h1) ...) conc))))
(define-syntax tc:rule-or
  (syntax-rules ()
    ((tc:rule-or t1 ...)
     (rule-or-helper (delay t1) ...))))

;; Define annotation rules and helpers
(define tc:infer-simple-datum-type
  (lambda (datum)
    (cond
      ((number? datum)  (tc:basic-type-expr 'number))
      ((char? datum)    (tc:basic-type-expr 'char))
      ((string? datum)  (tc:basic-type-expr 'string))
      ((boolean? datum) (tc:basic-type-expr 'boolean))
      ((symbol? datum)  (tc:basic-type-expr 'symbol)))))

(define tc:lambda-calc-annotate
  (lambda (pi e)
    (tc:set-current-subst-list! (list (tc:null-subst)))
    (tc:lambda-calc-annotate-lower pi e)))

(define tc:lambda-calc-annotate-lower
  (lambda (pi e)
    (cases tc:lambda-calc e
           (tc:le-self-evaluating
            (datum)
            (let ((t (tc:infer-simple-datum-type datum)))
              (tc:axiom (:- pi (: e t)))))
           (tc:le-varref
            (variable)
            (let ((lv (tc:new-logicalvar)))
              (tc:rule-if
               '() ;; no hypotheses here
               (lambda (ts)
                 (tc:env-bound? variable pi))
               (lambda (ts)
                 (tc:type-expr-bind lv (tc:env-value variable pi)))
               ;; ------------------------------------------------
               (:- pi (: e (tc:variable-type-expr lv))))))
           (tc:le-procedure-call
            (operator operand)
            (let ((rt           (tc:new-variable-type-expr))
                  (operand-type (tc:new-variable-type-expr)))
              (tc:rule (list
                         (:- pi (: operand operand-type))
                         (:- pi (: operator (tc:function-type-expr operand-type rt))))
                        ;;---------------------------------------------------------
                        (:- pi (: e rt)))))
           (tc:le-lambda-exp
            (formal body)
            (let ((rt (tc:new-variable-type-expr))
                  (formal-type (tc:new-variable-type-expr)))
              (tc:rule (list
                         (:- (tc:extend-env pi formal formal-type)
                             (: body rt)))
                        ;; ---------------------------------------------------
                        (:- pi (: e (tc:function-type-expr formal-type rt))))))
            )))
```

APPENDIX B. APPLICATION DERIVATION COMPLETION

During the discussion of how type rules are processed by the helpers in chapter 3 we made use of an example type derivation with the lambda calculus inference rules. This appendix completes the partial derivation given in this chapter. We last saw the derviation tree after the first top-level hypothesis had been recursively processed:

$$\dfrac{\dfrac{\dfrac{x : \tau_4 \in \Pi, x : \tau_4}{\Pi, x : \tau_4 \vdash x : \substack{\tau_4 \\ \tau_5}}}{\Pi \vdash (\lambda x : \substack{\tau_4 \\ \tau_1} . \, x) : \substack{\tau_4 \to \tau_5 \\ \tau_3 \to \tau_2}} \qquad \dfrac{\boxed{\vdots}}{\Pi \vdash n : \tau_3}}{\Pi \vdash (\lambda x : \tau_1 . \, x) \; n : \tau_2}$$

Next, we process the second hypothesis:

$$\dfrac{\dfrac{\dfrac{x : \tau_4 \in \Pi, x : \tau_4}{\Pi, x : \tau_4 \vdash x : \substack{\tau_4 \\ \tau_5}}}{\Pi \vdash (\lambda x : \substack{\tau_4 \\ \tau_1} . \, x) : \substack{\tau_4 \to \tau_5 \\ \tau_3 \to \tau_2}} \qquad \dfrac{n : \mathit{number} \in \Pi}{\Pi \vdash n : \substack{\mathit{number} \\ \tau_3}}}{\Pi \vdash (\lambda x : \tau_1 . \, x) \; n : \tau_2}$$

Logically, we propagate the substitution that $\tau_3 = \mathit{number}$ through the tree[8]:

$$\dfrac{\dfrac{\dfrac{x : \tau_4 \in \Pi, x : \tau_4}{\Pi, x : \tau_4 \vdash x : \substack{\tau_4 \\ \tau_5}}}{\Pi \vdash (\lambda x : \substack{\tau_4 \\ \tau_1} . \, x) : \substack{\tau_4 \to \tau_5 \\ \mathit{number} \to \tau_2}} \qquad \dfrac{n : \mathit{number} \in \Pi}{\Pi \vdash n : \mathit{number}}}{\Pi \vdash (\lambda x : \tau_1 . \, x) \; n : \tau_2}$$

[8]In reality, this propagation is immediate since a most general unifer is computed for expected and inferred types along the way. We deliberately show these steps to illustate a process similar to that followed when performing derivations by hand.

$$\dfrac{\dfrac{\dfrac{x : \mathit{number} \in \Pi, x : \mathit{number}}{\Pi, x : \mathit{number} \vdash x : \genfrac{}{}{0pt}{}{\mathit{number}}{\tau_5}}}{\Pi \vdash (\lambda x : \genfrac{}{}{0pt}{}{\mathit{number}}{\tau_1}.\ x) : \genfrac{}{}{0pt}{}{\mathit{number} \to \tau_5}{\mathit{number} \to \tau_2}} \qquad \dfrac{n : \mathit{number} \in \Pi}{\Pi \vdash n : \mathit{number}}}{\Pi \vdash (\lambda x : \tau_1.\ x)\ n : \tau_2}$$

$$\dfrac{\dfrac{\dfrac{x : \mathit{number} \in \Pi, x : \mathit{number}}{\Pi, x : \mathit{number} \vdash x : \mathit{number}}}{\Pi \vdash (\lambda x : \mathit{number}.\ x) : \genfrac{}{}{0pt}{}{\mathit{number} \to \mathit{number}}{\mathit{number} \to \tau_2}} \qquad \dfrac{n : \mathit{number} \in \Pi}{\Pi \vdash n : \mathit{number}}}{\Pi \vdash (\lambda x : \mathit{number}.\ x)\ n : \tau_2}$$

And finally,

$$\dfrac{\dfrac{\dfrac{x : \mathit{number} \in \Pi, x : \mathit{number}}{\Pi, x : \mathit{number} \vdash x : \mathit{number}}}{\Pi \vdash (\lambda x : \mathit{number}.\ x) : \mathit{number} \to \mathit{number}} \; \dfrac{n : \mathit{number} \in \Pi}{\Pi \vdash n : \mathit{number}}}{\Pi \vdash (\lambda x : \mathit{number}.\ x)\ n : \mathit{number}}$$

This completes the derivation.

BIBLIOGRAPHY

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*. Studies in logic and the foundations of mathematics. North-Holland Pub. Co., Amsterdam, 1958.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[FFF$^+$97] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. *Programming Languages: Implementations, Logics, and Program s*, 1292:369–388, September 1997.

[Fla04] Matthew Flatt. *PLT MzScheme: Language Manual (version 209)*, December 2004. Available from `http://download.plt-scheme.org/doc/209/html/mzscheme/index.htm`.

[FWH01] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, New York, NY, second edition, 2001.

[Gan77] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.

[JL96] Steven Jenkins and Gary T. Leavens. Polymorphic type-checking in scheme. *Computer Lanugages*, 22(4):215–223, 1996.

[Knu92] Donald E. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford University, 1992.

[LC05] Gary T. Leavens and Curtis Clifton. A type notation for Scheme. Technical Report 05-03, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 2005. Available by anonymous ftp from ftp.cs.iastate.edu.

[LCD05] Gary T. Leavens, Curtis Clifton, and Brian Dorn. A type notation for Scheme. Available from `http://www.cs.iastate.edu/~leavens/ComS342/docs/typedscm_toc.html`, April 2005.

[Lin93] Christian Lindig. STYLE: A practical type checker for Scheme. Informatik-Bericht 93-10, Technische Universitat Braunschweig, October 1993.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[PT98a] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, 1998.

[PT98b] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, 1998.

[Que03] Christian Queinnec. 23 things I know about modules for Scheme. In *Third Workshop on Scheme and Functional Programming, Pittsburgh, PA*, October 2003.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76. ACM, January 1989.

[Rod03] Daniel Rodríguez. Diseño y construcción de una herramienta para la generación automática de sistemas de tipos orientados a objetos (manual de usuario del framework TyS). Documento Interno 1022010, Department of Computer Science, Universidad de Oviedo, Oviedo, Spain, March 2003.

[Sch94] David A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.

[Wan87] M. Wand. Complete type inference for simple objects. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 37–44. IEEE, June 1987. Corrigendum in *Third Annual Symposium on Logic in Computer Science*, page 132, 1988.

[Wan89] Mitchell Wand. Semantic prototyping system (SPS) reference manual, version 1.4 (Chez Scheme). In ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lang/sps.tar.gz, Apr 1989.

[WC93] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. Technical Report COMP TR93-918, Department of Computer Science, Rice University, Houston, Texas, December 1993.

[WM98] Grant Wiggins and Jay McTighe. *Understanding by Design.* Association for Supervision and Curriculum Development, Alexandria, VA, 1998.