Simulated Overloading using Generic Functions in Scheme

by

Anthony M. Cox

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1997

Abstract

This thesis investigates extending the dynamically-typed, functional programming language Scheme, with simulated *overloading* in order to permit the binding of multiple, distributed definitions to function names. Overloading facilitates the use of an incremental style of programming in which functions can be defined with a base behaviour and then extended with additional behaviour as it becomes necessary to support new data types. A technique is demonstrated that allows existing functions to be extended, without modification, therefore improving code reuse.

Using the primitives provided by Scheme, it is possible to write functions that perform like the generic routines (functions) of the programming language EL1. These functions use the type of their arguments to determine, at run-time, the computation to perform. It is shown that by gathering the definitions for an overloaded function and building a generic routine, the language appears to provide overloading. A language extension that adds the syntax necessary to instruct the system to gather the distributed set of definitions for an overloaded function and incrementally build an equivalently applicable generic function is described.

A simple type inference algorithm, necessary to support the construction of generic functions, is presented and detailed. Type inference is required to determine the domain of an overloaded function in order to generate the code needed to perform run-time overload resolution. Some limitations and possible extensions of the algorithm are discussed.

Acknowledgements

Inevitably, during the process of acknowledging all the support and assistance given to the writer of a thesis, somebody is forgotten. Therefore, rather than mention specific names, I just want to thank everyone who contributed in some way to either this thesis or my attainment of the associated degree.

Special thanks must be given to Maryanne who has been on this journey with me for the past 5 years. Her love and support are a big part of the reason that these have been the best years of my life.

This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario and the University of Waterloo. This support is gratefully acknowledged.

Contents

1 Introduction **1**

1.1 Goal 1

1.2 Background 3

1.3 Terminology 6

2 Related Work **7**

2.1 Overloading 7

2.2 Related Concepts 9

2.3 Type Inference and Overloading 11

2.4 Type Inference in Scheme 14

2.5 Summary 17

3 DEFINE^{++}: The Language **19**

3.1 Overview 19

3.2 Overloading, Restrictions and Resolution 19

3.3 Explicit Domain Description 20

3.4 Domain Inference 21

3.5 Changes to Existing Scheme 24

3.6 Limitations and Complexities 25

3.7 New Functions 26

4 Design and Implementation **27**

4.1 Design Goals 27

4.2 General Description 28

4.3 Types in Scheme 31

4.3.1 Sources of Type Information 31

4.3.2 A Type System 32

4.4 Generation of Dispatch Code 33

4.5 Scoping and Dependency 35

5 Simple Type Inference in Scheme **39**

5.1 Overview 39

5.2 A Simple Type Inference Algorithm 40

5.2.1 Preprocessing 41

5.2.2 Constraint Generation 43

5.2.3 Constraint Solution 52

5.2.4 Termination Properties of the Algorithm 55

5.3 Discussion of the Algorithm 56

5.4 Recursion 58

6 Motivation and Application **61**

6.1 Motivation 61

6.2 An Example 63

6.3 Overriding from Overloading 64

6.4 The SmallScheme Library 66

6.5 Objects 68

6.6 The `Collection` Class 70

7 Conclusion **75**

7.1 Implementation Alternatives 75

7.2 Future Work 76

7.3 Summary 78

Bibliography **79**

List of Figures

1.1.1 Parametric Polymorphism in Scheme 2

1.2.1 Ambiguity Resulting from Overloaded Constants 3

1.2.2 Sum Coded as a Generic Function 4

1.2.3 Sum Coded Using Overloading 5

2.1.1 Type Classes in Haskell 8

2.2.1 Overloaded Methods in Java 10

2.2.2 Pattern Matching in ML 10

2.3.1 Example of Type Class Transformation 12

3.4.1 Two Identically Typed Versions of Plus 23

4.2.1 System Generated Code for Sum 29

4.2.2 Format of a Composite Definition 30

4.3.1 Definition of Types 32

4.4.1 Definition of DISPATCH 34

4.4.2 Overloading Based on a Functional Parameter 34

4.4.3 Order of Sub-definitions in a Composite Definition 36

4.5.1 Changing a Value Through Re-Definition 37

4.5.2 Illustration of a Type Dependency 37

5.2.1 The Function Length 40

5.2.2 Grammar for A-normalized Intermediate Form 42
5.2.3 A-normalization Algorithm 44
5.2.4 Results of Preprocessing Length 45
5.2.5 Recovery of Information from Type Artifacts 45
5.2.6 Relationships Between Variables and Types 46
5.2.7 An Example of Constraint Generation 48
5.2.8 Notation Used in Figure 5.2.9 49
5.2.9 Constraint Generation 50
5.2.10 Constraints Generated for Length 52
5.2.11 Definition of Type Intersection 53
5.2.12 Constraint Solution Algorithm 54
5.2.13 Constraint Solutions for Length 55
5.3.1 Inaccurate Type Generation 57
5.3.2 An Example of Type Inference 57

6.3.1 Simulating Overriding 65
6.4.1 SmallScheme Class Hierarchy 67

Chapter 1

Introduction

1.1 Goal

In order to increase the utility of a programming language, it is beneficial to improve its ability to abstract over both data and algorithms. Polymorphism is one mechanism language designers have used to achieve this goal. Cardelli and Wegner [12] have identified four distinct forms of polymorphism: parametric, inclusion (subtyping), overloading (incremental) and coercion. Each of these forms imparts different abstraction capabilities to a language.

Polymorphism exists only in languages with "a clear notion of both type and value" [12] since, by definition, a polymorphic language is one where "a value or variable (symbols) may have more than one type". Polymorphism, then, is a property of a language's type system,‘ and as a result, although an effective abstraction mechanism, is not always available in a programming language. The functional language Scheme, being dynamically typed, does not provide classical polymorphism since it lacks the necessary static type system. Scheme uses run-time type checking to ensure that the arguments passed to built-in, primitive functions match the domains of the primitives. Consequently, functions can be applied successfully to any arguments that do not cause a run-time type test to fail, irrespective of the types of the arguments.

Although Scheme is not considered to be parametrically polymorphic, it is possible to write functions that behave as though they were by avoiding the use of any primitives imposing restrictions on the functions parameters. Figure 1.1.1 provides examples of

```
;; Trivial parametric polymorphism
(define identity (lambda (x) x))

;; Polymorphic stack implementation
(define push (lambda (item stack) (cons item stack)))
(define top  (lambda (stack) (car stack)))
(define pop  (lambda (stack) (cdr stack)))
```

Figure 1.1.1: Parametric Polymorphism in Scheme

several such Scheme functions. Scheme exhibits a form of *natural polymorphism* since all functions have domains that are as general as possible within the limits of any run-time restrictions.

The natural polymorphism of Scheme has also been used to provide inclusion polymorphism. The current popularity of object oriented programming has generated several implementations of subtyping as extensions to the language [1] with the most successful and best known of these being the Meroon Object System [39]. For the most part, overloading and coercion appear to have been ignored as mechanisms useful for extending Scheme's expressiveness. This thesis concentrates on the former and has the goal of extending Scheme by using properties of the dynamic type system in order to provide the functionality of overloading.

The existence of heterogeneous lists, in which functions must perform differently on each element when mapped over the list, illustrates one useful application of overloading in Scheme. Using overloading, the function being mapped can adaptively select the appropriate behaviour needed for each different list element, as it is encountered, by using the type of the element.

Overloading can also be seen as an effective tool for improving code reuse. By writing functions that rely on overloaded operators to manipulate specific data types, these functions can be reused with new data types when the data type provides its own definition for the operator. Additional examples are presented in chapter 6, to give further evidence of the benefits provided by adding overloading to Scheme.

```
;; Overload "x"
(define x 1)
(define x "one")

;; Overload an output function
(define show (lambda (y) (if (string? y) (display y))))
(define show (lambda (y) (if (number? y) (display y))))

;; What should be displayed, 1 or "one"?
(show x)
```

Figure 1.2.1: Ambiguity Resulting from Overloaded Constants

1.2 Background

The following definition is based on those given by Cardelli and Wegner [12] and Aho, Sethi and Ullman [2]:

Overloading

> Overloading, also known as *incremental polymorphism*, is the allowing of multiple functional values to be bound to a symbol. The meaning, or binding to be used, for any particular instance of the symbol is determined from the symbol's context.

The process of selecting a binding based on the context (the location where a symbol occurs) is referred to as *overload resolution.* This process is static, if it is possible to determine from a compile time analysis the binding to use, or dynamic, if resolution is not determinable from analysis and must be delayed until run-time values are available.

The restriction to functional values is perhaps an arbitrary one. Cardelli and Wegner impose it since they class *value sharing*, where a symbol has multiple non-functional values, as a form of parametric polymorphism. This thesis only considers the overloading of functions for a different reason.

If an overloaded function were to be applied to an overloaded value, a potentially complex set of rules to successfully resolve the meanings of both the value and the function

```
(define sum
  (lambda (x y)
    (cond ((and (number? x) (number? y)) (+ x y))
          ((and (string? x) (string? y)) (string-append x y))
          (else (error "unknown input types for sum")))))
```

Figure 1.2.2: Sum Coded as a Generic Function

would be required. For example, in cases where possible ambiguity arises due to the use of an overloaded function and an overloaded value, should the function or the value be resolved first? In either case, the disambiguating rule rests on an arbitrary choice made by the language designer. The programming language Ada [50], allows the overloading of literals[1] and in doing so provides an example of the rule set needed to perform unambiguous resolution.

Since there is already an established trend in the literature to limit overloading to functions, it was decided to follow this trend to simplify the overload resolution rules, and to limit the effects resulting from decisions made to ensure consistent resolution. Figure 1.2.1 illustrates one instance of a situation requiring an arbitrary resolution rule when this restriction is removed. Wadler and Blott [52] also observe this phenomenon and comment that in a strongly typed language, additional type information, beyond that which can be inferred, may be required to resolve ambiguity caused by unrestricted overloading of constants.

Intuitively, an overloaded function can be perceived as one with several different possible behaviours, where the type of the arguments are used to determine which behaviour to utilize. Activity closely matching this intuitive view already occurs in Scheme, and is illustrated in figure 1.2.2. The programming language EL1 [54] provides a construct known as a *generic routine* that acts like this version of `sum`. From this example, it can be seen that Scheme's dynamic type system permits the language to provide the generic construct of EL1.

While a generic function[2] does not satisfy the definition of overloading, since the function name is only given a single binding, it does provide the same behaviour. When examined in the context of its call-site, a generic function becomes indistinguishable from

[1] A literal is a member of an enumerated type.

[2] A routine in EL1 is equivalent to a function in Scheme.

```
(define   sum (lambda (x y) (error "unknown input types for sum")))
(define++ sum (lambda (x y) (string-append x y)))
(define++ sum (lambda (x y) (+ x y)))
```

Figure 1.2.3: Sum Coded Using Overloading

an overloaded one. For both generic and overloaded functions, the context surrounding an instance of a function call infers the behaviour the function provides. When the function itself is examined, it becomes evident that the types of the arguments bound to the parameters (x and y) determine the appropriate behaviour. From a programming stance, the significant difference between a generic function and an overloaded one is that the definition for the overloaded function may be decomposed and distributed throughout a program, while the definition for the generic function must occur in a single location. This characteristic advantageously allows overloading to be used to incrementally extend the domain of an existing function.

The function `sum` can be seen coded using overloading[3] in figure 1.2.3. Examining this function reveals how this can be referred to as incremental polymorphism. A basic definition is given for `sum`, which is then extended in an incremental fashion. Comparing the generic and the overloaded version of `sum` it can be seen that the generic function gathers the definitions of the overloaded version and merges them together using the `cond` construct. From a different perspective then, overloading is a facility that allows the branches of a conditional to be distributed throughout a program.

To achieve the goal of the thesis, the language is extended with the addition of a set of macros that build the code of figure 1.2.2 when given the code from figure 1.2.3. These macros gather the individual definitions for an overloaded function and build a generic version of the function. The generic version can be used identically to the overloaded version making the construction transparent to users of the language. By using macros, the extension can be added to any version of Scheme regardless of the versions implementation details.

[3]The keyword `define++` will be described in chapter 3. It can be treated as identical to `define` in this example.

1.3 Terminology

Up to now, the word *type* has been used rather loosely, relying upon an intuitive meaning of the word. To be more precise, a type is a set of values. For identification, these sets are commonly identified with a name such as `boolean`. Thus, though `boolean` is referred to as a type, the reference is actually to the set of values, `{#t, #f}`, the type name represents. A type may be infinite, such as the set named `number`, representing all numeric values in Scheme, or it may be finite, as is the case with the previously enumerated type `boolean`. In this thesis, type names are presented in a `typewriter` font and traditional set notation is used to describe and manipulate the sets represented by type names.

Referring again to the overloaded definition of `sum` given figure 1.2.3, each (component) definition is referred to as a *sub-definition* and the definition of `sum`, which is built from these component definitions, as a *composite definition*.

In Scheme, a *predicate* is traditionally a function returning a boolean value [14]. The term should not be confused with the idea of a *type predicate* introduced by Wadler and Blott [52]. Type predicates are implicit constraints on type variables imposed by the use of overloaded operators. It should be noted that the boolean symbol for true (`#t`) returned by a predicate is distinct from the more general concept of *truth* in Scheme. Any value, except the boolean symbol for false (`#f`), is considered as being true. Functions like `member`, which do not always return a `boolean` value, are therefore usable as predicates since the values they return satisfy the definition for truth.

Chapter 2

Related Work

2.1 Overloading

Two kinds of overloading exist within programming languages: built-in and user-defined. Built-in overloading refers to overloaded functions provided by the language specification, such as the overloading of the operator + for both real and integer types. In user-defined overloading the user is given the ability to select which symbols are overloaded and to provide the various definitions for these symbols. Some languages, such as Ada [50] and C++ [49], provide both varieties, while others, such as FORTRAN 77[1] [6] and Pascal [31], provide only built-in overloading. It should not be inferred that all languages support some form of overloading since some, for instance BLISS [58], provide no overloading of any form. In general, built-in overloading adds limited expressiveness to a language and so, for the remainder of this thesis, only user-defined overloading is considered.

Since the concept of polymorphism is closely tied to that of type, overloading traditionally occurs in strongly typed languages where it has the characteristic of being statically resolvable at compile time. Three strongly typed programming languages providing overloading, C++, Ada, and Haskell [30], are examined before considering other related concepts.

In C++, overloading is limited to functions. A function can only be overloaded if each definition has a unique type which differs in more than the return type. Most, but not all, operators can also be overloaded. A complex set of rules is used to determine if two

[1]FORTRAN 90 provides both built-in and user-defined overloading.

```
:: Declare a type class named Math
class Math a where
  (==), (+)

:: Make Int an instance of Math
instance Math Int where
  (==) = eqInt
  (+)  = addInt

:: Make Float an instance of Math
instance Math Float where
  (==) = eqFloat
  (+)  = addFloat

:: Use class to add implicit parameter inference
double   :: Num a => a -> a
double x =  x + x
```

Figure 2.1.1: Type Classes in Haskell

types differ. For example, the types `const T` and `volatile T` are indistinguishable for some type `T` while `const T&`[2] and `volatile T&` are distinguishable. Overload resolution is performed statically during compilation and is based upon the "best match" for the actual arguments. Much of the complexity of choosing a match results from the automatic conversions (coercions) which C++ inherited from C.

In Ada, it is permissible to overload literals, operators and functions, but not variables or constants. The return type of a function may be used in overload resolution making overloading in Ada less restrictive than in C++. To perform resolution, the type, mode[3] and name information from the arguments must identify exactly one possible function to apply.

Haskell differs from the previous two languages by being functional instead of imperative in nature. Overloading is provided through *type classes*, which group types that share

[2]`&` as a suffix makes a type a reference or alias in C++.

[3]Function parameters have modes `in`, `out` and `in out` to specify whether they are readable or writable.

common functions. Any function in a type class becomes overloaded if more than one type is declared as an instance of that type class. In the example of figure 2.1.1 a class `Math` is declared that describes types providing the operators `==` and `+`. The types `Int` and `Float` are then declared as instances of `Math` overloading `==` and `+`.[4] Type classes can inherit from other type classes, where a subclass extends its superclass by providing additional functions, and can be considered analogous to the classes of object oriented programming but different in that there is no notion of an internal mutable state. Wadler and Blott [52] have demonstrated that type classes are a type safe extension of the Hindley-Milner type discipline implemented by Haskell.

2.2 Related Concepts

Since Scheme is a dynamically typed language, it is incapable of providing overloading as modeled by the previous three languages. The dynamic nature of the type system prevents overload resolution from being consistently performed statically. It is therefore beneficial to examine other concepts bearing a strong relation to overloading, particularly those with a dynamically resolved component.

Overriding, introduced as part of object oriented programming, is comparable to overloading and occurs when a subclass re-implements a method provided by one of its superclasses. Rouaix [43] documents the similarity between overloading and overriding that can be found when overloading is resolved at run-time (dynamically) instead of statically. The difference between statically resolved overloading and dynamically resolved overriding can be seen in Java [24], where a two phase resolution procedure is used for method invocations.

Java, a descendant of C++, differs by not providing a simple mechanism to express function abstraction. The object oriented character of the language requires all functions to be implemented as class methods thus intertwining overloading and overriding. To perform overload resolution the arguments passed to a method are used at compile time to create a call-site type signature, which is then used at run-time, to invoke the appropriate method using "dynamic method lookup". Figure 2.2.1 provides an example of an overloaded and overridden method in Java. In the class `RealPoint`, the method `move` is overloaded to have parameters of either type `float` or type `int`, while overriding the implementation of its superclass, `Point`.

[4] `eqInt`, `addInt`, `eqFloat` and `addFloat` are primitive operations in Haskell.

```
class Point {
  int x = 0, y = 0;
  void move (int dx, int dy) {
    x += dx; y += dy;
  }
} /* end class Point */

class RealPoint extends Point {
  float x = 0.0f, y = 0.0f;
  void move(int dx, int dy) {
    x += (float) dx; y += (float) dy;
  }
  void move (float dx, float dy) {
    x += dx; y += dy;
  }
} /* end class RealPoint */
```

Figure 2.2.1: Overloaded Methods in Java

```
exception Join

fun join (a, b)           = a andalso b
|   join (a:int, b:int)   = a + b
|   join (a::[], b::[])   = (a ^ b)::[]
|   join (_,_)            = raise Join
```

Figure 2.2.2: Pattern Matching in ML

Pattern matching, as it is found in Standard ML [38], is another technique akin to overloading. With pattern matching, the structure of the argument types is used to perform the selection of the appropriate behavior to provide. In the case of Standard ML, this selection occurs dynamically during execution. Examining a simple ML function in figure 2.2.2, it can be seen that each pattern is treated like a separate definition for the function `join`. The patterns given describe the following argument types:

1. Two boolean types.
2. Two integer types. (The parameter declarations are needed to resolve the built-in overloading on the operator `+`.)
3. Two single element lists of strings.
4. Any other types not specified above.

Pattern matching differs from overloading by requiring that all function definitions with the same name be sequentially located instead of distributed. Pattern matching has been implemented as an extension to Scheme by Wright and Duba [57].

The programming language EL1 [54] provides a facility called a *generic routine* that also uses the run-time type of the arguments to select the code to apply to the arguments. This technique is used in other, more recent languages, such as with the `TYPECASE` construct in Modula 3 [11]. Generic routines, like pattern matching, do not provide the incrementality available from overloading. As was demonstrated, dynamically typed languages like Scheme and Lisp, can effectively provide generic routines using conditional expressions and their built-in primitive functions. This capability is exploited and generic routines used to simulate overloading in Scheme.

2.3 Type Inference and Overloading

From a theoretical perspective, there has been a significant amount of research done on type systems supporting overloading. However, since Scheme does not require a user to provide any form of type declaration, a type system implemented to support overloading must also allow the inference of any needed types.

In modern functional languages, the most influential research on type inference is the development of the algorithm W [17], which statically determines types according

```
data MathD a = MathDict (a -> a -> bool) (a -> a -> a)

eql (MathDict e a) = e
add (MathDict e a) = a

mathDInt       :: MathD Int
mathDInt       = MathDict eqInt   addInt

mathDFloat     :: MathD Float
mathDFloat     = MathDict eqFloat addFloat

double         :: NumD a -> a -> a
double dict x =  (add dict) x x
```

Figure 2.3.1: Example of Type Class Transformation

to the Hindley-Milner type discipline [29, 37]. The original algorithm was only designed to handle parametric polymorphism and did not consider any other varieties. Several languages, most notably SML and Haskell, have been designed around this type system.

Wadler and Blott [52], as part of the implementation of Haskell, added overloading to W using type classes.[5] Their technique is expressed as a compile time transformation that converts a program with overloading into an equivalent program without overloading, which is typable under W. This approach allows type classes to be added to existing languages through the addition of a preprocessor.

When type classes are used, the functions that are members of the type class are translated into record field selectors. Instances of the class can then have their member functions bundled together into a record referred to as a *method dictionary.* Any function using an overloaded function is given an extra parameter to allow the appropriate dictionary to be passed in. This technique has an effect similar to universally quantifying the member functions and then restricting the possible instantiations of the quantifier to the set of method dictionaries. An example of this transformation can be seen in figure 2.3.1 where the code of figure 2.1.1 has been converted.

In the example, a new data type `MathD`, with the type constructor `MathDict`, is defined.

[5]Type classes were previously described in section 2.1.

All types, which are instances of the class `Math`, then use this constructor to build a dictionary (e.g., `mathDFloat` and `mathDInt`). Functions, such as `double`, that use the members of the type class, are modified to allow the appropriate dictionary to be passed in as a parameter. A function that is a member of the type class can therefore be replaced with the application of a "selector" to the dictionary. This substitution can be seen in `double` where the function `+` is replaced with the first member of the dictionary, selected using `add`.

Wadler and Blott's system, while effectively used in Haskell, is not decidable. Although it is possible to assign a type to an expression involving overloading, it is impossible to determine if the type is valid. More particularly, though well-typed programs can be typed under W, the Wadler and Blott algorithm also allows a type to be inferred for ill-typed ones. Smith [47] gives an example using the expression `true + true` for which a type can be inferred, even though `+` is undefined for booleans. Earlier work by Smith [46] demonstrated that type-checking an expression with unrestricted overload instantiation, as is the type class approach, is reducible to the Post Correspondence Problem and is thus undecidable.

In parallel with the work by Wadler and Blott, Stefan Kaes [33] developed a similar extension to W. His parametric overloading is also a form of restricted universal quantification based on parameter passing. Kaes's system does not use a mechanism such as type classes to group types that share the same overloaded functions but it imposes the restriction that the type scheme of an overloaded function have at most one type variable. Duggan, Cormack and Ophel [20] characterize the type system of Wadler and Blott as a generalization of Kaes's system.

Volpano and Smith [48], restrict the system of Wadler and Blott in order to create a decidable type system. Their initial restriction is to disallow mutual recursion among overloaded functions. This restriction is proven to be insufficient to remove undecidability. Building on this, they add additional restrictions until a decidable type system is achieved. The key restriction imposed is that for a type variable, in the type of an overloaded function, all possible instantiations of the variable have a *unique* outermost type constructor. Duggan et al. [20] provide a good summary of the Volpano and Smith restrictions.

Geoffrey Smith [47] later investigated the addition of both overloading and subtyping to the Hindley-Milner type system. His result is an extension to type theory that permits overloading while preserving the inference of *principal types* by extending the type system with *constrained universal quantification.* Principal types are a useful characteristic of the algorithm W and are defined as a “best” type that captures all possible types for a program. Constrained universal quantification associates a set of constraints with each universal quantifier such that quantified variables are only allowed to take instantiations satisfying the constraint set. Previous research on extensions to W ignored the principal type property.

Recent work by Duggan, Cormack and Ophel [19, 20] on *kinded types*, provides another approach to extending W with parametric overloading. By relaxing some of the restrictions imposed by Volpano and Smith with respect to overloaded recursive functions and adding some to non-recursive functions they improve the expressibility of the system while maintaining decidability. To reduce the overall limitations of the theory, the type system is extended with *kinds* (sets of types). This system maintains the requirement of *discriminativity* where all instantiations of a type variable in an overload template have a unique outermost type constructor.

Type dependent parameter inference proposed by Cormack and Wright [15] provides a system combining type inference and overload resolution but is not based on the Hindley-Milner approach. Specifically, this approach differs in requiring that function abstractions (`lambda` expressions) be explicitly typed. This approach recognizes the dependence that functions have on the overloaded operators used in their implementation. By parameterizing these functions over the overloaded operators they can be made more general. It is then easy to use these general functions by implicitly inferring the instantiations needed for the parameters.

2.4 Type Inference in Scheme

After examining overloading with respect to type inference, the next issue is the performance of type inference in a dynamically typed language such as Scheme, which is not an easy task since function parameters may now be instantiated with any value that does not cause a run-time test to fail. This generalization has the effect of generating types (sets of values) containing large numbers of unrelated values. These sets are commonly represented as the union of existing types like `number` and `string`.

Ma and Kessler [36] classify type inference approaches as either functional or imperative. The functional approach refers to algorithms based on the previously mentioned Hindley-Milner type discipline for functional languages. The imperative approach is geared towards assignment-based imperative style languages. Jones and Muchnick [32] developed the original algorithm used in imperative languages, but their algorithm has been superseded by a generalization of it by Kaplan and Ullman [34]. The imperative approach has been used in type inferencers for functional languages and so both approaches are examined here.

Current research supports *soft-typing*, which is the optimization of a program by using type information to remove unneeded run-time checks. During soft-typing, conservative approximations for types are permissible and only cause (potentially) unnecessary type checks to be left in a program. Various algorithms have been presented based on tree-grammars [53], abstract interpretation [3, 45, 51], extensions of algorithms for statically-typed languages [35, 55] and constraint satisfaction [27, 21]. Some of these approaches are examined.

The work of Olin Shivers [45] uses a transformation to continuation passing style (CPS) to create a version of a Scheme program with explicit control and data flow. He then creates a non-standard abstract semantics to build sets of values for each *computational quantity*[6] identified by CPS conversion. However, complexities in the control and data flows prevent the abstract semantics from building accurate sets. To solve this problem, he adds an additional "reflow" semantics, which is comparable to a separate analysis of each control flow path. The approach taken in this thesis bears some similarities to this work, but differs by following control flow, as opposed to data flow, to identify types.

Recent work by Flanagan and Felleisen [21] on set-based analysis for Scheme has also had some influence on this thesis. Their approach, based on Nevin Heintze's set-based analysis of ML [26], generates constraints and applies them interactively. Like Shivers, their algorithm uses a simplified intermediate form, namely A-normal form [22]. The system is designed to type complete programs and only returns the type associated with the result obtained by evaluating the program.

The SoftScheme system by Andrew Wright [56, 55] is based on the work of Cartwright and Fagan [13], which extends the Hindley-Milner system to a dynamically typed language. In effect, SoftScheme does the opposite of the overloading extension developed here. Using the similarity between the generic nature of Scheme functions and overloading, SoftScheme uses an encoding technique designed by Rèmy [40] to reduce run-time overloading to parametric polymorphism. Circular, instead of traditional, unification [42] is used to

[6]Shivers defines a computational quantity as anything that can be considered as having a type.

properly unify recursive type expressions. The type system of SoftScheme provides both recursive types and disjoint union types.

Alex Aiken and Brian Murphy [3] advocate an abstract interpretation based on the operational semantics of their algorithm for the language FL [8]. Using regular trees, they generate sets of type expressions to describe types in the language. Aiken considers the approach limited in value since it takes exponential time and abandons it to consider more efficient approaches.

In his later work, Aiken [4, 5] switches to a constraint approach for type inference. His key difficulty is in finding solutions to the sets of constraints. In general, it can be shown that unless restrictions exist, the sets may be unsolvable. In cases when constraints can be solved, it is difficult to do so efficiently. Wright comments on Aiken's work in [56], considering it a "strong type system", although he believes it is at times less precise for simple functions and its complexity makes it slower than SoftScheme. The type system has two interesting components: constraint types and conditional types. Constraint types are constraints upon the instantiation of type variables in types reported as part of a result type. For example, there are no recursive types since a list can be expressed by the type: $\alpha \subseteq$ `cons`$(\beta,\alpha) \cup$ `null` **and** $\alpha \supseteq$ `cons`$(\beta,\alpha) \cup$ `null` where $\alpha \subseteq \beta$ indicates that the type α is constrained by the type β. As opposed to merging the types for each branch of an `if` expression, as is done in other approaches, conditional-types push the conditional into the actual type value. Conditional types were first introduced by Reynolds [41]. One issue concerning their system is its rejection of some sound (type error-free) programs by the algorithm as a side-effect of the restrictions needed to guarantee solvability of the constraint sets.

Fritz Henglein [27, 28] has studied type inference in dynamically-typed languages for a different purpose. He develops a technique for merging statically and dynamically typed languages. Type inference is used to identify sites where *coercions* are required to ensure run-time type safety. Like Aiken, Henglein also uses a constraint satisfaction approach. The successful translation of Scheme into ML has been performed by relying on a library of ML functions to perform run-time type checking. Henglein does not attempt to accurately type all Scheme since his concern is in locating potential type errors of Scheme programs under the ML type system. Using a conservative approach to type inference, unresolvable values are coerced into the type "dynamic", which is then tested at any run-time use-site to see if it contains a usable value.

A common technique used to represent programs is as a parse tree, with operations as nodes and values as leaves. A nodes "context" is the portion of the tree immediately above it. Thus, when moving down the tree, before a node or leaf is encountered, the context can be used to determine an expected type. For example, with an addition node, each branch under the node can be expected to have a type of `number`. Upward movement in the tree allows functions to be assigned types using the types of the operands. As an alternative to the algorithm W, Kaplan and Ullman [34] iteratively traverse a parse tree in both directions until they achieve stable types for every node and leaf.

The Nimble type inferencer for Common Lisp-84 [9] is based on the Kaplan-Ullman method of fixed point iteration. It is difficult to find documentation on the Nimble inferencer since the source code is the property of the Nimble Computer Corporation. The Nimble inferencer does not attempt to type functions, only variables and constants, since its goal is to expose optimization opportunities related to storage management. During type inference, both upper and lower bounds are kept for each object, allowing iteration to be halted when either the bounds meet or differ by some predetermined amount.

Similar to Nimble, the TICL (Type Inference for Common Lisp) [36] type inferencer also uses fixed-point iteration. TICL is a preprocessor for Common Lisp programs, which adds type declarations to offer compilers better opportunities for optimizing storage management. Ma and Kessler appear to have developed TICL in parallel with Baker's development of Nimble since neither author references the other, while the two resulting inferencers are similar in operation.

2.5 Summary

The approach to overloading used in this thesis types function abstractions incrementally as they are defined. Since the specification for Scheme permits the definition of functions with unbound symbols, the type inference algorithm used must be able to cope with these "open" systems. This characteristic, resulting from the syntax, permits mutually recursive functions to be defined independently. With many of the approaches to type inference that were examined, it was unclear as to how to adapt them to deal with an open system.

Considering the multitude of approaches to type inference in dynamically typed languages, it is not evident which is best suited for use in this application. The Hindley-Milner type discipline has successfully been extended to handle overloading and has been used in a dynamically typed language, but these two objectives have not been achieved simultaneously.

The other approaches to dynamic type inference offered no obvious solutions and each demonstrated limitations. As a result, ideas were borrowed from many of these systems and a simple control-flow oriented type inference algorithm based on Kaplan-Ullman fixed point iteration was designed specifically for this application.

Chapter 3

DEFINE^{++}: The Language

3.1 Overview

This chapter serves as a reference manual and as an introduction to the DEFINE^{++} language extension for Scheme developed as part of this thesis. The choice of the name is meant to convey the idea of overloading through incremental definitions by combining the well known C++ increment operator '`++`', with the Scheme keyword `define` used to indicate a top-level definition. There are actually two components to DEFINE^{++} — the syntactic extensions to Scheme needed to provide overloading, and the SmallScheme library that is described in chapter 6.

3.2 Overloading, Restrictions and Resolution

A function is overloaded if, at the top or global level, it has several different definitions.[1] For each call of an overloaded function, the run-time types of the arguments are used to select the sub-definition to apply. Overloading is restricted to top-level functions only, but existing primitive operations, such as `+` or `display`, may be overloaded without limitation.

In order to determine which sub-definition to apply for a particular function call, the types associated with the arguments for the call are examined to see if they satisfy the domain of one of the sub-definitions. In Scheme it is impossible to determine the exact

[1]Section 1.3 defines the term *sub-definition* to describe the component definitions of an overloaded function.

types of a function's arguments until the arguments are supplied during execution. For this reason, selection of the appropriate function sub-definition to use must be deferred until run-time. To perform this selection (overload resolution) DEFINE^{++} provides two mechanisms for describing the domain of each sub-definition. The first allows the programmer to explicitly describe the domain using any arbitrary Scheme expression, while in the second the system attempts to infer the domain from a simple abstract analysis of the function body.

Overload resolution is easy when the domain of only one sub-definition satisfies the types of the arguments. However, if more than one sub-definition is applicable a "disambiguating rule" must be applied. Sub-definitions are ordered with respect to the sequence they were entered into the system. In the event that more than one sub-definition applies the most recently entered is used. Should no sub-definition apply, the first definition entered (least recent) is applied, regardless of its domain. This least recent sub-definition is consequently referred to as the *default definition*. Should the domain of the default definition not match the types of the arguments a run-time error occurs.

The choice of a disambiguating rule for overload resolution was motivated by the desire to have overloading used in an incremental fashion. Later definitions extend the domain of a function, or in some cases alter the function's behaviour for a subset of its existing domain. Therefore, the rule allows programmers to establish the most general behavior for a function first, then refine or extend it as desired.

3.3 Explicit Domain Description

The first technique available for describing a functions domain is to have the programmer explicitly describe it using an arbitrary Scheme expression, which is accomplished with a `lambda++` definition. This technique is the more powerful of the two presented since it allows for very sophisticated domains to be described. It is also the only technique a programmer needs to learn in order to begin using overloading.

Syntax: (`lambda++` *formals domain body*)
Description: *formals* and *body* are like their counterparts in `lambda` while the additional component *domain* is an arbitrary Scheme expression that is used as a predicate and applied to the run-time arguments of the function abstraction. If the `lambda++` expression

is not bound to a top-level (global) symbol or is bound to a unique, new symbol (the symbol is not overloaded), *domain* is ignored and the expression is treated as a regular `lambda` expression. If the expression is bound as the second or subsequent definition of a top-level symbol, *domain* is used to determine if *body* is evaluated. The *domain* expression of each sub-definition is evaluated, from most recently defined to least recently defined, until one evaluates to true. If *domain* yields any value other than `#f`, *body* is evaluated and no further *domain* expressions are evaluated. Any side-effects occurring from the evaluation of *domain*, even if it evaluates to `#f`, are applied. The *domain* expression of the default sub-definition is never evaluated.

Example:

```
(define F (lambda (x y) (list x y)))
(define F (lambda++ (x y) (member x y) y))
```

In this example, `F` is overloaded and is applicable to either domains of lists containing a specified member, as seen in the second (most recent) sub-definition, or to domains of any two arbitrary values. The lack of restrictions implied by the first (default) sub-definition allows the second sub-definition to provide an alternative result for a sub-set of the overall function domain. The function `F` can be considered, after overloading, to be equivalent to the following:

```
(define F (lambda (x y)
            (cond ((member x y) y)
                  (else (list x y)))))
```

From another perspective, `lambda++` can be considered as an indirect approach to pattern matching. The *domain* clause of the abstraction can be any Scheme expression returning a boolean value and describing the desired form of the parameters. This is little more than using the existing syntax of Scheme to describe a pattern instead of extending the syntax, as was done by Wright and Duba [57].

3.4 Domain Inference

An alternative to having the programmer supply the domain to use for each sub-definition, of an overloaded function, is to have the system attempt to infer the sub-definitions domain. Overloading using this method only occurs when requested through the use of the `define++` syntax.

Syntax: (`define++` *name expr*)
Description: `define++` acts like `define` when it is used to bind non-functional expressions, such as constants, or when it is used to bind the first (default) function sub-definition to a name. When it is used to bind the second and subsequent function sub-definitions to a name, the domain of the sub-definition is inferred[2] and then used to generate "dispatch code". This dispatch code acts in exactly the same manner as the programmer supplied domain expression of the `lambda++` syntax. The dispatch code of each sub-definition is evaluated, from the most recent until the least recent sub-definition, until it evaluates to a true value at which time the appropriate function body is evaluated. If no true result has been obtained prior to checking the default sub-definition, the body of the default definition is used regardless of its domain and even if it results in a run-time error. The alternative short-hand syntax for definitions:

    (define++ (*name formals*) *body*)

which expands to:

    (define++ *name* (lambda *formals body*))

is also supported.

In many instances, as can be seen in example 3.4.1, it is not possible to accurately infer the domain of a function using only its body. From the two definitions of `+` given, it is impossible to infer that the first definition applies when `x` and `y` are three element lists with the tag `cart` as the first element and the second definition applies when `cart` is replaced by the tag `polar`. To overcome this problem assertions have been added to the language using the `assert` keyword described below.

Syntax: (`assert` *bool-expr expr*)
Description: This expression evaluates *expr* and returns the result of this evaluation. The *bool-expr* is not evaluated and is used solely during the domain inference process, so no side-effects resulting from the evaluation of *bool-expr* can occur. It is asserted that *bool-expr* evaluates to true allowing the programmer to use standard Scheme code to add type information. The *expr* clause does not have an implied `begin`.

[2]This inference is performed using the algorithm described in chapter 5

```
;; Constructors for User-Defined types
(define make-polar (lambda (x y) (list 'polar x y)))
(define make-cartesian (lambda (x y) (list 'cart x y)))

;; Cartesian Sum
(define + (lambda (x y)
  (list (first x)
        (+ (second x) (second y))
        (+ (third x) (third y))))

;; Polar Sum
(define + (lambda (x y)
  (list (first x)
        (sqrt (+ (square (second x)) (square (second y))))
        (sqrt (+ (square (third  x)) (square (third  y))))))
```

Figure 3.4.1: Two Identically Typed Versions of Plus

The new keyword `assert` is required to indicate that the actual sub-definition should not include the *bool-expr* since it is used only to aid type inference. In many instances, assertions are little more than having the programmer specify (indirectly) the predicates to use for dispatch, and consequently, assertions can be viewed as a weak form of `lambda++`. The difference between the two is that `lambda++` allows the description of any input domain while `assert` is limited to describing domains representable by the type system.

Example:

```
        (define++ F (lambda (x) (assert (= x 0) "zero")))
```

In this example, the function `F` ignores its argument and returns the constant (string) value of `"zero"`. However, the assertion, that the parameter (x) is equal to the value `0`, causes to type `(-> 0 "zero")` to be inferred for `F` instead of the more accurate type `(-> any "zero")`. The use of an assertion permitted type information, not inferable from the body of the function abstraction, to be "discovered".

No attempt is made to check the consistency of the assertion, putting the responsibility on the programmer to provide valid assertions. Should an inconsistent or erroneous assertion be made, the system may either generate an incorrect type or may be unable to generate any type. The dispatch code generated for an untypable expression always evaluates to true, effectively serving to remove any previously entered sub-definitions and turning the untyped expression into a new default sub-definition.

3.5 Changes to Existing Scheme

Definitions are still accomplished using the `define` special form, however its behaviour has been slightly altered. As well as performing the binding of a value to a symbol, `define` infers and stores the type of the symbol. Type inference is performed when the symbol is not being overloaded in order to improve the types inferred for overloaded symbols. When a globally bound symbol is used in an overloaded function, its value can be looked up and used to provide additional type information during type inference for the function. Should some error occur that prevents a type from being generated for the value, a type of `void` is assigned. There is no effect on the definition process when this occurs.

The original binding of the special form `define` is preserved using the syntax `defun`. In the event of system failure, it is possible to rebind `define` to its original value to allow the system to be reset. It is also possible to bind symbols using `defun` but in doing so, no type is maintained by the language extension. For this reason, the use of `defun` should be avoided.

No other changes have been made to the base language. Existing Scheme programs continue to execute without change, unless, by coincidence there is a top-level binding to the name of one of DEFINE^{++} 's internally used functions. To prevent this, programmers should avoid using symbols with the prefix '`dpp:`'. All internally used functions of the extension are prefixed in this manner.

As part of the overloading process a "name-mangling" is done on previous function names that become overloaded. For any function `F`, the names `F:1` ... `F:`n are also bound where n is the number of sub-definitions of `F`. This is not done if `F` has only one sub-definition (i.e., is not overloaded). In order to prevent the meaning of a program from changing, programmers should be aware of this name-mangling to avoid accidentally rebinding one of the renamed sub-definitions.

3.6 Limitations and Complexities

Limitations inherent in the current type system often prevent the desired dispatch code from being generated. These limitations can be summarized as:

1. Basic types (`number`, `string`, `boolean` etc.) can only be inferred for parameters if the parameter is used in some context requiring the use of a specific basic type.
2. All parameters which are functions (i.e., have a type of `(-> ...)`) are considered to have the same type.
3. `cons` types must have an inferable basic type in one of the fields to be differentiable.
4. Lists can not be inferred and are treated as either a `cons` type or `null` depending on use.
5. All uninferable parameters are given the special type `any` to which any actual argument may be bound.

For example, it may be necessary to overload a binary function of two parameters, `x` and `y`, such that new behaviour occurs when `x` and `y` are equal length lists and have the same last element. Since the type system provides no facility for representing a recursive type such as a list, there is no possibility of inferring "patterns" such as this even using assertions. In other words, the system can not infer types that result in the generation of the dispatch code:

```
(and (list? x) (list? y)
     (= (length x) (length y))
     (equal? (last x) (last y)))
```

To circumvent this problem, the programmer must provide the actual dispatch code using the previously described `lambda++` syntax.

When a `lambda++` is used within a `define++`, the inference process is used and the explicit domain description of the `lambda++` is treated as an assertion. This can be summarized by considering:

`(define++` *name* `(lambda++` *formals domain expr*`))`

to be equivalent to:

`(define++` *name* `(lambda` *formals* `(assert` *domain expr*`)))`

3.7 New Functions

Two new functions have been added to the language. The first, `type`, allows the system to generate dispatch code and is available to the user in order to aid in the task of program development.

Syntax: `(dpp:type` *expr*`)`
Return Value: An expression representing the type of *expr*
Description: Using the type recovery algorithm described in chapter 5, *expr* is typed and a type, as described in figure 4.3.1, is returned. In the event the expression is untypable, the type `void` is returned.

After typing, the extension stores the types of all symbols bound in the top-level scope in a type environment. It is possible to access this environment to determine the type of a symbol using the function `type-of`.

Syntax: `(type-of` *symbol*`)`
Return Value: An expression representing the type bound to *symbol* or `unbound` if the symbol is not bound in the top-level scope.
Description: The type returned matches the definition of types in figure 4.3.1.

Chapter 4

Design and Implementation

4.1 Design Goals

Before describing the design and operation of the language extension, it is beneficial to present the design criteria in order to provide the rationale for many of the decisions made. There were two goals used to guide language design with the first motivating the second.

1. Existing Scheme programs must execute without change.
2. Minimal new syntax.

To be classed as an extension to Scheme, the package must add new capabilities while maintaining the existing syntax and semantics. Thus, existing programs must execute in the extended language in exactly the same way as they would in the original language. This goal was met except for programs containing functions whose names, by coincidence, match the keywords of the new syntax. This motivates the next design goal; minimization of new syntax to mitigate naming conflicts.

To preserve existing semantics, it was necessary to limit the amount of syntax introduced. This criteria was met since only three constructs — `lambda++`, `define++` and `assert` — were added to the language. It is also hoped that by minimizing the amount of new syntax the language would be easy to learn and use.

Other less formal design goals existed, but it is hard to concretely express ideas such as "ease of use" and "expressiveness". Ideally, it is desirable to have very simple and easy to use constructs capable of expressing very complex ideas. It is hoped that users will find that DEFINE^{++} satisfies this ideal, which can only be measured as feedback is obtained from the use of the package.

4.2 General Description

The addition of overloading to the language changes the actions that must take place at binding time. When a symbol is uniquely (initially) defined, no special handling is required, and as a result, Scheme can bind the function to the symbol in the normal manner. Definition, though it can be considered as unchanged in this case, has been slightly altered so that it types each expression and stores these types in a type environment.

When a function is specified as overloaded, on the second and all subsequent definitions, the binding construct must replace the current definition with a merged form of it and the new sub-definition. This extension is done by determining the appropriate Scheme predicate functions (e.g., `number?`, `string?`, `list?` etc.) to be applied to the arguments in order to make the run-time selection of the appropriate sub-definition. The run-time determination of the code to execute is known as *dynamic dispatch.* Every run-time call of an overloaded function results in dispatch occurring, to perform overload resolution, for the function.

Figure 4.2.1 shows the exact code generated by the macro `define++` when `sum` is defined using overloading, as was seen in figure 1.2.3. More generally, the construction of a composite definition can be characterized according to the format in figure 4.2.2. Note, it is shown later how to generate the dispatch code referenced in this figure.

It should be noted that the original sub-definitions are not incorporated into the composite definition. Should the composite definition need to be rebuilt, it is impossible to retrieve each sub-definition's *expr* component because Scheme does not provide any method of accessing the original "code" component of a closure. By maintaining each sub-definition as a separate closure it becomes an easy matter to regenerate the composite definition, during incremental extension, since no code needs to be retrieved. Extension, due to overloading, involves adding a new branch to the beginning of the `cond` expression every time a new sub-definition is added. The name for each sub-definition is produced by suffixing the name of the composite definition with a colon and an integer representing the order of its definition, beginning with one. Renaming can be seen illustrated in figure 4.2.1 where the sub-definitions for the function `sum` are defined as `sum:1`, `sum:2` and `sum:3`.

```
(define sum:1
  (lambda (x y) (error "unknown input types for sum")))
(define sum:2
  (lambda (x y) (string-append x y)))
(define sum:3
  (lambda (x y) (+ x y)))

(define sum
  (lambda (var1 var2)
    (cond ((and (number? var1) (number? var2))
           (sum:3 var1 var2))
          ((and (string? var1) (string? var2))
           (sum:2 var1 var2))
          (else
           (sum:1 var1 var2)))))
```

Figure 4.2.1: System Generated Code for Sum

Knowing how to generate a composite definition leads to the most complex issue and the majority of the implementation: the determination of the appropriate dispatch code for each sub-definition. When the user provides the dispatch code using the keyword `lambda++` this issue is trivial; however, when the `define++` mechanism is used it becomes far more difficult because of the need to determine the sub-definitions domain.

When the `define++` form is used, some kind of abstract analysis of each sub-definition is needed to determine the *types* for which correct execution is possible. More precisely, what is needed is a function which, given a syntactically and semantically correct Scheme expression, returns a type expression accurately describing the expression. The term "syntactically and semantically correct" describes Scheme expressions that execute without generating run-time errors. The type expression returned must be in a form suitable for generating dispatch code using the predicate functions available in Scheme. Such a type system is described later in this chapter. The analysis of a (type) declaration-free program in order to assign types to the functions and variables is traditionally known as *type inference*.

For some overloaded function:

$$
\begin{aligned}
\mathrm{F}_1 &= \texttt{(lambda (x}_1\ \texttt{...}\ \texttt{x}_m\texttt{)}\ \mathit{expr}_1\texttt{)} \\
\mathrm{F}_2 &= \texttt{(lambda (x}_1\ \texttt{...}\ \texttt{x}_m\texttt{)}\ \mathit{expr}_2\texttt{)} \\
&\ddots \\
\mathrm{F}_n &= \texttt{(lambda (x}_1\ \texttt{...}\ \texttt{x}_m\texttt{)}\ \mathit{expr}_n\texttt{)}
\end{aligned}
$$

where m $\geq$ 0, n $\geq$ 2

The composite definition is:

```
F_comp = (lambda (x_1 ... x_m)
           (cond (δ_n     F:n)
                 (δ_{n-1} F:n-1)
                      ⋱
                 (else F:1)))
```

where δ_i represents the dispatch code for $expr_i$, 2 $\leq$ i $\leq$ n

and `F:`j is bound to F_j, 1 $\leq j \leq$ n

Figure 4.2.2: Format of a Composite Definition

4.3 Types in Scheme

4.3.1 Sources of Type Information

Before describing any possible type systems, it is advantageous to examine the sources of type information, or type artifacts, available in Scheme.

1. Applications of primitive operations.
   Example: `(+ x y)`
   The above application of `+` indicates that `x` and `y` are both of type `number`. It should be noted that primitives provide information about both parameters and return values. The type of a primitive is obtained by looking up the name of the function in a type environment maintained by DEFINE^{++}.

2. Conditional expressions (alteration of control flow).
   Syntactic forms such as: `if`, `cond`, `case`, `and`, `or`
   Example: `(if (number? x) (/ x y) (f x))`
   There are four pieces of type information in the above example. The first two are that the "guard" clause `(number? x)` returns a true result in the "then" clause and a false result in the "else" clause. The other two are the knowledge that `x` is of type `number` when the guard is true and is of some type distinct to `number` otherwise.

3. Assertions.
   Example: `(assert (number? x) (/ x y))`
   The `assert` expression indicates that, in the expression `(/ x y)`, the variable `x` has a type that is a subset of the domain of `number?`.

As previously mentioned, assertions are a syntactic addition of DEFINE^{++} used to add extra (declarative) information for the purpose of improving type inference. Although necessary when using inference based dispatch code generation, it is desirable to limit the need for assertions. The requirement to handle implementation details such as this takes away time that programmers should be using on high-level (algorithmic) issues. As well, adding assertions negates one of the significant advantages of Scheme, with respect to ease of use, over a language such as C, where the user is forced to manually declare types. For these reasons, it is important that the type inference algorithm make the best possible use of any existing type artifacts and only require assertions when available type information is insufficient.

Empty Type:	`void`
Universal Type:	`any`
True Type:	`true`
Unknown Type:	`unspecified`
Primitive Types:	`number`, `string`, `char`, `boolean`, `vector`, `symbol`

Constant Types:

$\quad c \in \texttt{number} \cup \ldots \cup \texttt{symbol} \cup \{\texttt{null}\}$

Constructed Types:

$\quad$ `(cons` $type_1$ $type_2$`)`

$\quad$ `(or` $type_1 \ldots type_n$`)`

$\qquad$ where $type_i$ is distinct from $type_j$ if $i \neq j$

$\quad$ `(->` $type_1 \ldots type_m$ $type_{m+1}$`)` where $m \geq 0$

Figure 4.3.1: Definition of Types

4.3.2 A Type System

Scheme does not have a static type system. As mentioned in section 1.1, it does have a theoretical type system defined in the (partial) formal semantics included in [14], but this is only used to assist in the presentation of a denotational semantics for the core language and selected primitive operations. Thus, using these semantics as a basis and considering the predicates available to perform dispatch, the type system shown in figure 4.3.1 has been designed.

The type `void` occurs as the result of a Scheme expression that generates a run-time error, while the type `unspecified` is returned when a function has no return type, such as the input-output primitives (e.g., `display`, `newline`, `read`, etc.) of Scheme. The universal type `any` is the set of all types except for the type `void`. To satisfy the formal language specification [14], there are two types for "true". Since all values, except the symbol `#f` (false), satisfy the definition of truth in Scheme, the type `true` represents the set `any - {#f}`. The symbol for true `#t`, which is a member of the type `boolean`, is a part of, but does not represent, the previously mentioned type `true`.

It is significant that the type system described here does not express any recursive types, such as lists. This is a deliberate omission discussed more fully in section 5.4.

4.4 Generation of Dispatch Code

When types have been inferred, using the procedure described in the next chapter, they are used to generate the appropriate dispatch code for function sub-definitions.

Since only functions are overloaded, all sub-definitions have the form:

$$\texttt{F}_i \texttt{ = (lambda (x}_1 \ldots \texttt{x}_n\texttt{)} < \mathit{expr} >_i\texttt{)}$$

which have a type of:

$$\texttt{(->} < \mathit{type} >_1 \ldots < \mathit{type} >_n < \mathit{type} >_{return}\texttt{)}$$

Thus, dispatch code for F_i is of the form:

$$\texttt{(and } \textsc{dispatch}[\texttt{x}_1\texttt{,} < \mathit{type} >_1] \ldots \textsc{dispatch}[\texttt{x}_n\texttt{,} < \mathit{type} >_n]\texttt{)}$$

where the definition of the meta-function DISPATCH is found in figure 4.4.1.

Note that, as indicated, the dispatch code for a sub-definition is generated differently from that for any parameters to the sub-definition that have a function type. This rule prevents a user from overloading a function based on the type of a parameter that is itself a function, as is seen in figure 4.4.2. The lack of existing Scheme predicates, other than `procedure?`, to identify functions prevents the generation of any useful dispatch code. In example of figure 4.4.2, the parameter `f` is a function in both sub-definitions and so dispatch code of the form `(procedure? f)` is generated for `f`. Dispatch can not occur based on the type of `x` since it is a `number` in both sub-definitions. Thus, the lack of existing Scheme predicates results in the generation of identical dispatch code for the two definitions, motivating the above rule.

The actual dispatch code generated by the macros may differ slightly from that described in figure 4.4.1 because of the occurrence of some optimizations such as replacing `(and (equal? x '()) #t)` with `(null? x)`. The optimization which occurs is very simplistic in nature and is limited to improving the code generated for a single sub-definition. No attempt is made to improve the overall efficiency of the composite definition by optimizing the complete dispatch code.

The meta-function DISPATCH previously described generates the dispatch code for one sub-definition, however it does not indicate how the dispatch code for each sub-definition

DISPATCH:$[\mathit{name}, \mathit{type}] \Rightarrow \mathit{scheme\text{-}expr}$

DISPATCH[`x, void`] ::= *undefined*

DISPATCH[`x, any`] ::= `#t`

DISPATCH[`x, p`] ::= `(p? x)`

DISPATCH[`x, c`] ::= `(equal? x c)`

DISPATCH[`x, (cons` τ_1 τ_2`)`]

    ::= `(and (pair? x)` DISPATCH[`(car x),` τ_1] DISPATCH[`(cdr x),` τ_2]`)`

DISPATCH[`x, (or` τ_1 `...` τ_n`)`]

    ::= `(or` DISPATCH[`x,` τ_1] ... DISPATCH[`x,` τ_n]`)`

DISPATCH[`x, (->` τ_1 `...` τ_m τ_{m+1}`)`] ::= `(procedure? x)`

where τ = type
    x = variable
    p $\in$ {`number, string, char, vector, boolean, symbol`}
    c $\in$ {p | p $\in$ primitive types} $\cup$ {`null`}

Figure 4.4.1: Definition of DISPATCH

```
;; This definition has type: (-> (-> string any) number any)
(define apply-f (lambda (f x) (f (number->string x))))

;; This definition has type: (-> (-> number any) number any)
(define++ apply-f (lambda (f x) (f (+ x 1))))
```

Figure 4.4.2: Overloading Based on a Functional Parameter

is ordered in the composite definition. Recall the disambiguating rule presented in section 3.2: *in the event that more than one sub-definition applies, the most recently defined is used.* Using this rule, the following convention can be adopted:

> *The order of the sub-definitions in the composite definition inversely matches the order of their definition.*

That is, the most recently defined sub-definition is first, the next most recent is second and so on. With this approach, the default behaviour for the function comes from the oldest sub-definition, which is correct according to the specification given in chapter 3. Figure 4.4.3 illustrates the construction of a composite definition from three sub-definitions. In the `cond` construct of the composite definition, the inverse order of the sub-definitions is easily seen.

Validity

> A sub-definition is *valid* if, for the arguments provided by the call-site, the dispatch code for the sub-definition evaluates to `true`. For non-overloaded functions, a definition is always valid.

As the defined ordering indicates, the most recently encountered sub-definition for which the dispatch code evaluates to true is used. This ordering rule maintains and extends the existing semantics of Scheme. Currently, a symbol is rebound if a second (or subsequent) definition occurs for an already bound symbol. In the extension, this behaviour is maintained if the most recent definition is *valid* for the call-site (in current Scheme this must be the case or a run-time error occurs). However, if it is not valid, all other sub-definitions are considered until either a valid sub-definition is found, or all sub-definitions are examined and none are found to be valid forcing the default definition to be used regardless of its domain.

4.5 Scoping and Dependency

Scheme is a statically (lexically) scoped language, which means a function call is evaluated using the environment the function is defined in. However, the primitive `define` is imperative in nature, and changes the value stored in a memory location without changing any bindings to that location, as is illustrated in the Scheme program of figure 4.5.1.

For the sub-definitions *evaluated in the order given*:

```
;; Definition 1 (renamed to enumerate:1)
(define enumerate
  (lambda (x) (error "unknown input type for enumerate")))

;; Definition 2 (renamed to enumerate:2)
(define++ enumerate (lambda (x) (assert (number? x) x)))

;; Definition 3 (renamed to enumerate:3)
(define enumerate
  (lambda++ (x)
    (list? x)
    (if (null? x) 0 (+ 1 (enumerate (cdr x))))))
```

The following composite definition is generated:

```
(define enumerate
  (lambda (var1)
    (cond ((list? var1)   (enumerate:3 var1))
          ((number? var1) (enumerate:2 var1))
          (else           (enumerate:1 var1)))))
```

Figure 4.4.3: Order of Sub-definitions in a Composite Definition

```
;; Define some things in global scope.
(define x 1)
(define show-x (lambda () (display x)))

;; The following returns the value 1.
(show-x)

;; Change the value of x.
(define x 2)

;; The following returns the value 2 stored in x
;; and NOT the 1 stored in x when show-x was defined
(show-x)
```

Figure 4.5.1: Changing a Value Through Re-Definition

```
;; Define a pair of functions.
(define sum (lambda (x y) (+ x y)))
(define double (lambda (x) (sum x x)))

;; Overload sum. Type of double changes!
(define sum (lambda (x y) (string-append x y)))
```

Figure 4.5.2: Illustration of a Type Dependency

In the function `show-x`, the reference to `x` in `display` is bound to the location named `x` in the global scope. The second definition of `x` changes the value stored in the memory location, but does not affect the actual binding of the symbol `x` to that location. Therefore, the second call to `show-x` uses the binding to the location established at its definition time, even though the value of the location has changed. This behaviour can have significant effects in the presence of overloading, as the short program of figure 4.5.2 demonstrates.

As the example illustrates, overloading can change the type signature of previously defined functions. This will be referred to as a *type dependency*. In the example, the type of the function `double` is dependent upon the type of the function `sum`. Changing the type of `sum` changes the type of `double`, which may then change the type of other functions dependent upon `double`.

In order to maintain an accurate type for each functions, it is necessary to store a dependency list for every function defined. At the time of a functions definition, it is added to the dependency list of all the functions it calls. Whenever a function is overloaded and the overloading causes the type of the function to change, all functions in its dependency list must be re-typed. Any changes in the type of a function results in the subsequent re-typing of all the functions on its dependency list, until no more re-typing is necessary.

When a function, `f`, calls a function, `g`, which directly or indirectly calls `f`, a recursive cycle is formed. To ensure this does not lead to an infinite cycle in the re-typing algorithm, the name of the function being overloaded is stored and re-typing stops when that function is encountered. This approach can lead to overly general types for the members of a recursive cycle, however, when functions are typed independently, precise types are impossible to achieve.

Chapter 5

Simple Type Inference in Scheme

5.1 Overview

In order to automate the generation of dispatch code, the use of available type information to determine types (as previously described in figure 4.3.1) for each sub-definition is investigated. This investigation results in the design and construction of a control-flow based type inference algorithm for Scheme.

It is important to note that perfect type inference is unnecessary. The extension works if overly general types are returned for a sub-definition (or component thereof). Should the inference algorithm return a type of any, the new sub-definition overrides any existing sub-definitions and reproduces the existing behaviour of Scheme. Overly general types simply limit the effectiveness of overloading and do not cause erroneous or unpredictable behaviour.

As an example, if a sub-definition of type `(-> number number boolean)` is inferred to have the type `(-> any any boolean)`, dispatch code of `(and #t #t)` is generated. This dispatch code causes DEFINE^{++} to mimic the existing semantics of Scheme, since it always evaluates to true causing the associated sub-definition to be used. The constant selection of a sub-definition is effectively the same as rebinding the symbol to the sub-definition. This behaviour is not considered "incorrect" but does link the effectiveness of the `define++` syntax to the accuracy of the type inferencer.

As a result of the above limitation, it is vital that the extension report its activities so that the user can predict which sub-definition is used for a particular function instantiation.

```
(define length
  (lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l))))))
```

Figure 5.2.1: The Function Length

The reporting of the type used also permits the programmer to effect improved inference by strategically adding additional declarative information using `assert`.

A simple type inference algorithm, as is provided in the implementation, is described, followed by a discussion of the limitations and possible extensions of the algorithm. Particular attention is paid to the subject of recursive types, which are not handled by the simple algorithm.

5.2 A Simple Type Inference Algorithm

The algorithm described is not complete in that it does not generate any form of recursive type or consider the "non-functional" aspects of Scheme, such as assignment. What is presented is intented to provide effective overloading while simultaneously offering an easily extended framework for the future implementation of additional features.

The inference algorithm can be broken down into three sequential stages: preprocessing, constraint generation and constraint solution. To clarify and illustrate the operation of the algorithm, the composite results of applying each stage sequentially to the function given in figure 5.2.1 are given. The example is a simple, one-parameter function, which calculates the length of a null terminated list and returns the result as an integer. Although the algorithm does not generate recursive types, a recursive function is furnished, as an example, to demonstrate the algorithms handling of recursion.

The use, to create dispatch code, of the algorithm results in its application to solitary Scheme expressions, since the top level function `type` is called on the *expression* component of a `define` construct.[1] Overloaded functions are therefore typed incrementally since `type` is applied to each sub-definition as it is defined.

[1]The syntax for a top level definition is: (`define` *variable expression*).

5.2.1 Preprocessing

In order to simplify the actual type inference process, the source code undergoes the following transformations in the preprocessing stage:

- All variables are renamed with unique names.
- Syntax is simplified and standardized.
- The expression is A-normalized.
  (This is a transformation very similar to conversion to continuation passing style.)

The activities of this stage simplify the implementation of the second and third stages of the algorithm by reducing source programs to a normalized subset of Scheme.

During renaming, all variables are replaced with new, unique names. The renaming process adheres to all of Scheme's scoping rules and thus generates different names for identically named variables within different scopes. This removes the need to identify variables with respect to scoping restrictions in future stages. New variable names are integer values prepended with the character `$`.

Simultaneously, during renaming, all of the source code is standardized to replace complex syntactic constructs with simpler ones and to transform constructs with alternative forms of syntax into a common form. The following standardizations are performed:

1. Replacement of `let*` and `let` with equivalent `letrec` constructs.
2. Conversion of alternative syntax for definitions into a standard format with explicit `lambda`'s.
3. Replacement of `and` and `or` with appropriate and equivalent constructs using `if` and `letrec`.
4. Replacement of `cond` and `case` with equivalent `if` expressions.
5. Insertion of any missing `else` clauses for `if`, `cond` and `case` expressions. The function `(void)` returning an unspecified value is used for any missing clauses.
6. The insertion of `begin` in all situations where it is implied, such as in `lambda` expressions, `cond` cases and `letrec` bodies.

$$
\begin{array}{lll}
\mathit{EXPR} & ::= & \mathit{VAR} \\
 & | & (\texttt{let}\ (\mathit{VAR}\ \ \mathit{CONST})\ \mathit{EXPR}\ ) \\
 & | & (\texttt{let}\ (\mathit{VAR}\ \ (\texttt{if}\ \mathit{VAR}\ \ \mathit{EXPR}_{true}\ \mathit{EXPR}_{false}))\ \mathit{EXPR}\ ) \\
 & | & (\texttt{let}\ (\mathit{VAR}\ \ (\texttt{lambda}\ (\mathit{EXPR}_1\ \ldots\ \mathit{EXPR}_n)\ \mathit{EXPR}_{body}))\ \mathit{EXPR}\ ) \\
 & | & (\texttt{let}\ (\mathit{VAR}\ \ (\mathit{FUN}\ \mathit{EXPR}_1\ \ldots\ \mathit{EXPR}_n))\ \mathit{EXPR}\ )
\end{array}
$$

FUN ::= VAR | primitive procedure

CONST ::= constant

VAR ::= variable

Figure 5.2.2: Grammar for A-normalized Intermediate Form

7. Replacement of `do` with a locally defined recursive function using `letrec`.
8. Replacement of all quoted lists with explicitly constructed lists using `cons`.
9. Decomposition of `cadr` and similar functions into their `car` and `cdr` components.

The final action during preprocessing is to transform the standardized representation into A-normal form using A-normalization [22]. This transformation is very similar to conversion to continuation passing style (CPS) [7], and recently Sabry and Wadler [44] have shown CPS conversion and A-normalization generate results that are Galois reflections[2] of each other. A-normalization conveniently names all intermediate values in computations in addition to further simplifying the source code by reducing it to the form represented by the grammar in figure 5.2.2. The advantage gained from the extra work of this transformation is that it becomes far easier to solve constraint sets when there are no missing constraints inferred by anonymous intermediate values. It is notable that the transformation generates an intermediate representation of a Scheme program and not executable Scheme code. In particular, the semantics of `let` in the intermediate are not those of `let` in Scheme, but are instead those of `letrec`.

Both CPS conversion and A-normalization have been used as part of type inference algorithms implemented for Scheme; CPS by Shivers [45] and A-normalization by Flanagan

[2] Galois theory is beyond the scope of this thesis and so is not detailled here. Readers are urged to refer to [44] for a more detailed analysis of the relationship between CPS conversion and A-normalization.

and Felleisen [21]. The latter was chosen because the added overhead of continuations is not required by the type inference algorithm being described.

The normalization code is the same as that in Appendix 1 of [22], apart from the extension for the `begin`[3] construct and the naming of constants. The implementation uses continuations in a technique pioneered by Danvy and Filinski [18], however, figure 5.2.3 reproduces this algorithm in terms of a direct transformation. Preprocessing the `length` function yields the code given in figure 5.2.4. For readability, the machine generated unique variable names (`$1`, `$2`, ...) have been replaced by ones considerably more descriptive in nature.

5.2.2 Constraint Generation

Description of Constraints

The next stage of the algorithm generates a set of equality constraints for each control flow path. Before describing the process of generating these constraints, the nature of a constraint is examined and a definition given.

Although the term "constraints" is used, the term "equality" is also suitable. A constraint, which is always associated with a variable, serves two roles. The first is as a bound on the set representing the type of the variable. The second is to describe equivalences between the type of the associated variable and the type, or component thereof, of some other variable. Both these roles are now illustrated using examples.

Since all computational quantities are named by A-normalization, all type artifacts are associated with variables. The type information available from a particular artifact represents the type of a variable at some point during execution. However, the type information may be more general and indicate a larger set of values than the variable could actually have.

Figure 5.2.5 describes a preprocessed version of a function which adds `1` to its argument. For the variable `$3` occurring in the application of `+`, given by `(+ $2 $3)`, it is possible to infer that `$3` has type `number`. The `let` binding to the constant `1` actually indicates that the type of `$3` is in fact the value 1 and so the type `number` can be seen as overly general and represents a larger set of values than the variable is ever assigned. As this

[3] `begin` is a sequencing construct used primarily to obtain side-effects.

$$\text{A-NORMALIZE}[\mathit{expr}] = \mathcal{A}[\mathit{expr},\ \nu,\ \nu]$$

where $\mathcal{A}[\mathit{expr},\ \mathit{name},\ \mathit{body}] \Rightarrow$ *normalized-expr* is defined:

$\mathcal{A}[\mathrm{v}, \text{name}, \text{body}] ::= \mathrm{v}$

$\mathcal{A}[\mathrm{c}, \text{name}, \text{body}] ::=$ `(let` (name c) body)

$\mathcal{A}[(\mathrm{E}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n), \text{name}, \text{body}]$

$\quad ::= \mathcal{A}[\mathrm{E}, \nu, \mathcal{A}[(\nu\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n), \text{name}, \text{body}]]$ where $\mathrm{E} \neq \mathrm{f}$

$\mathcal{A}[(\mathrm{f}\ \mathrm{v}_1\ \ldots\ \mathrm{v}_m\ \mathrm{E}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n), \text{name}, \text{body}]$

$\quad ::= \mathcal{A}[\mathrm{E}, \nu, \mathcal{A}[(\mathrm{f}\ \mathrm{v}_1\ \ldots\ \mathrm{v}_m\ \nu\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n), \text{name}, \text{body}]]$ where $\mathrm{E} \neq \mathrm{v}$

$\mathcal{A}[(\mathrm{f}\ \mathrm{v}_1\ \ldots\ \mathrm{v}_m), \text{name}, \text{body}] ::=$ `(let` (name (f $\mathrm{v}_1\ \ldots\ \mathrm{v}_n$)) body)

$\mathcal{A}[($`if` $\mathrm{E}_g\ \mathrm{E}_t\ \mathrm{E}_f), \text{name}, \text{body}]$

$\quad ::= \mathcal{A}[\mathrm{E}_g, \nu, \mathcal{A}[($`if` $\nu\ \mathrm{E}_t\ \mathrm{E}_f), \text{name}, \text{body}]]$ where $\mathrm{E}_g \neq \mathrm{v}$

$\mathcal{A}[($`if` $\mathrm{v}\ \mathrm{E}_t\ \mathrm{E}_f), \text{name}, \text{body}]$

$\quad ::=$ `(let` (name (`if` v $\mathcal{A}[\mathrm{E}_t, \nu_1, \nu_1]\ \mathcal{A}[\mathrm{E}_f, \nu_2, \nu_2]$)) body)

$\mathcal{A}[($`lambda` $(\mathrm{v}_1\ \ldots\ \mathrm{v}_n)\ \mathrm{E}), \text{name}, \text{body}]$

$\quad ::=$ `(let` (name (`lambda` ($\mathrm{v}_1\ \ldots\ \mathrm{v}_n$) $\mathcal{A}[\mathrm{E}, \nu, \nu]$)) body)

$\mathcal{A}[($`letrec` $((\mathrm{v}\ \mathrm{E}_1))\ \mathrm{E}_2), \text{name}, \text{body}]$

$\quad ::= \mathcal{A}[\mathrm{E}_1, \mathrm{v}, \mathcal{A}[\mathrm{E}_2, \text{name}, \text{body}]]$ where $\mathrm{E}_1 \neq \mathrm{c}$ or v

$\mathcal{A}[($`letrec` $((\mathrm{v}\ \mathrm{v}^c))\ \mathrm{E}), \text{name}, \text{body}]$

$\quad ::=$ `(let` ($\mathrm{v}\ \mathrm{v}^c$) $\mathcal{A}[\mathrm{E}, \text{name}, \text{body}]$) where $\mathrm{v}^c = \mathrm{v}$ or c

$\mathcal{A}[($`begin` $\mathrm{v}_1\ \ldots\ \mathrm{v}_m\ \mathrm{E}\ \mathrm{E}_1 \ldots \mathrm{E}_n), \text{name}, \text{body}]$

$\quad ::= \mathcal{A}[\mathrm{E}, \nu, \mathcal{A}[($`begin` $\mathrm{v}_1\ \ldots\ \mathrm{v}_m\ \nu\ \mathrm{E}_1 \ldots \mathrm{E}_n), \text{name}, \text{body}]]$ where $\mathrm{E} \neq \mathrm{v}$

$\mathcal{A}[($`begin` $\mathrm{v}_1\ \ldots\ \mathrm{v}_n), \text{name}, \text{body}] ::=$ `(let` (name v_n) body)

where E = expression

ν = a new (unique) variable

v = variable or primitive operation

c = constant

f = symbol bound to a function

Figure 5.2.3: A-normalization Algorithm

```
(let (length (lambda (x)
     (let (bool (null? x))
          (let (if-expr (if bool
                            (let (zero 0) zero)
                            (let (cdr-x (cdr x))
                                 (let (result (length cdr-x))
                                      (let (one 1)
                                           (let (sum (+ one result))
                                                sum))))))
               if-expr))))
     length)
```

Identification of new variables introduced by A-normalization:

`bool`	= result of `null?`
`if-expr`	= result of `if`
`zero`	= constant value 0
`cdr-x`	= cdr field of `x`
`result`	= result of recursive call to `length`
`one`	= constant value 1
`sum`	= sum of 1 and result of recursive call

Figure 5.2.4: Results of Preprocessing Length

```
(let ($1 (lambda ($2)
            (let ($3 1)
                 (let ($4 (+ $2 $3)) $4)))) $1)
```

Figure 5.2.5: Recovery of Information from Type Artifacts

```
(let (head (lambda (x)
             (let ($1 (car x)) $1)))
     head)
```

Figure 5.2.6: Relationships Between Variables and Types

simple example demonstrates, constraints represent local upper bounds on the type of the associated variable. Recall that this was the first role that constraints filled. To explain the second role of constraints (type equivalences) the following definition is required.

Enhanced Type

> A type is an *enhanced type* if it can be described using the type system defined in figure 4.3.1 enhanced with the addition of program variables.

For example, `(cons $2 any)` is an enhanced type representing a pair having the type found for the variable `$2` (after constraint solution) in its first field and the universal type `any` in its second field.

When type constructors such as `cons` and `lambda` or destructors such as `car` and `cdr` are used in programs, additional information is recovered indicating type relationships between variables. Constraints instrument the transfer of type knowledge both into and out of constructed types by linking the variables associated with the constructed type and with its component types. The type constructors of the type system are used to describe the movement of type information into a constructed type while the general purpose type destructor (`fieldOf` *var n*) describes the movement of type information out of the n^{th} field of the type of *var*.

Figure 5.2.6 contains a simple Scheme function that returns the "head" (first element) of a list formed using `cons` types. The variable `$1` was added by preprocessing to explicitly name the functions return value. In the figure, it is possible to see that the type of the return value `$1` is the same as the type of the `car` field of the parameter `x`. This is evident from the binding of the `car` of `x` to `$1` in the `let` expression. To represent this relationship `$1` is constrained by `(fieldOf x 1)` and `x` by `(cons $1 any)`.

There are three different forms that constraints can take. The first moves type information between variables that, at some point during program execution, have equal bounds on their types. The second serves the role of a type destructor and moves information out of constructed types into its components. The last acts as a type constructor and moves information into a constructed type from its components.

1. `(=` *variable*`)`
   This indicates that all constraints upon *variable* also apply to the variable being constrained. It does not indicate any constraints upon *variable*.

2. `(fieldOf` *variable* N`)`
   This indicates that *variable* is a constructed type and that the variable associated with the constraint is constrained by the Nth field of that type. These constraints serve as type destructors.

3. *enhanced types*
   When a type constructor is used, an enhanced type is suitable for constraining the variable to which the type constructor is bound. Enhanced types can be seen as indicating both a bound upon a variables value and a transfer of type information to and from the variables named by the component types.

Figure 5.2.7 illustrates the constraints generated for a simple Scheme function, which takes no arguments and returns the constant `3`. The constraint set generated is organized as an environment with entries of the form:

variable: *constraints*

For the variables, `$1` to `$5`, the constraints generated indicate the following:

- `$1` is a function of no arguments with a return value that is constrained by the type of `$5`.
- `$2` is the constant value `3`.
- `$3` is the constant value `6`.
- `$3` is a `cons` type with its `car` field constrained by the type of `$2` and its `cdr` field constrained by the type of `$3`.
- `$5` is constrained by the type of the first field of `$4`.

Given the Scheme expression:

```
(lambda () (car (cons 3 4)))
```

Preprocessing generates:

```
(let ($1 (lambda ()
           (let ($2 3)
                (let ($3 6)
                     (let ($4 (cons $2 $3))
                          (let ($5 (car $4)) $5)))))) $1)
```

Which generates constraints:

```
$1:  (-> $5)          $4:  (cons $2 $3)
$2:  3                $5:  (fieldOf $4 1)
$3:  6
```

Figure 5.2.7: An Example of Constraint Generation

To summarize, constraints can be used to describe local bounds on the set of values (type) a variable can represent at run-time, as well as describing locations where type information is obtained during constraint solution.

Generation of Constraints

One of the unique aspects of the type inference algorithm is its generation of a type for each control flow path. Instead of trying to merge the type information from both branches of an `if` expression, the function containing the expression is typed twice, once considering the guard expression evaluates to `true`, and once considering it returns `#f`, assuming the value of the guard is not statically determinable.

Generating constraints is not a complex procedure. Since expressions in the intermediate form are tree-like in nature, it is a simple matter of traversing the tree, in a depth-first manner, generating a constraint set for each branch of the tree. Any side-effects from imperative style assignment functions are ignored.

Before presenting the meta-function used to describe the constraint generation procedure the context in which it is used must be clear. Since constraints are generated for

Let γ = a constraint

v = a variable

c = a constant

E = a constraint environment of the form $\{(v_1, \gamma_1) \ldots (v_m, \gamma_m)\}$ where $m \geq 0$

$E^{(n)}$ = a set of n constraint environments $\{E_1 \ldots E_n\}$ where $n \geq 0$

Define FINALVAR as:

FINALVAR[`(let` (v *expr'*) *expr*)] ::= FINALVAR[*expr*]

FINALVAR[v] ::= v

Figure 5.2.8: Notation Used in Figure 5.2.9

each control flow path, one of the parameters is a set of constraint sets. Initially this set contains a single empty constraint set. When a branch in the control flow is reached, such as an `if` expression, *each* constraint set is duplicated and specialized for a specific branch of the conditional expression. Constraint generation then continues down each branch, using a different copy of the constraint set in each branch. After each branch is constrained the resulting "sets of constraint sets" are unioned together. The other parameter is the expression, in the intermediate form, to be constrained.

Figure 5.2.8 presents the notation used in figure 5.2.9 to provide the formal definition of the constraint generation function:

$$\mathcal{C}[\mathit{expr}, \{E_1 \ldots E_n\}] \Rightarrow \{E_1 \ldots E_m\};\ m,n \geq 0$$

This function describes the constraints generated for each expression in the intermediate form where $\mathcal{C}$ is initially applied with one empty constraint set to the complete A-normalized intermediate representation of the function being constrained.

Resolution of overloaded functions during inference is treated as though each potential instance of the function were a separate control flow path. This traversal can be readily seen by comparing the rule for constraining an overloaded application (rule 7d) with the one for an `if` expression (rule 8). Both rules extend the set of constraint environments by causing duplication of each existing set and then specializing, through the addition of different constraints, each copy of the set.

For example, assuming that the function `+` is overloaded for both the type `number` and

(1) $\mathcal{C}[\texttt{v},\ \mathrm{E}^{(n)}]$

(a) If `v` is locally bound

$::= \mathrm{E}^{(n)}$

(b) If `v` is bound at the top level to τ

$::= \{\mathrm{E}_i \cup \{(\texttt{v},\ \tau)\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}$

(2) $\mathcal{C}[\texttt{(let (v c) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}] ::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v},\ \texttt{c})\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(3) $\mathcal{C}[\texttt{(let (v (lambda (v}_1 \texttt{ ... v}_n\texttt{) } \mathit{expr}_{body} \texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v},\ \texttt{(lambda v}_1 \ldots \texttt{ v}_n\ \textsc{finalvar}[\mathit{expr}_{body}]\texttt{))}\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(4) $\mathcal{C}[\texttt{(let (v (cons v}_1\texttt{ v}_2\texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v},\ \texttt{(cons v}_1\texttt{ v}_2\texttt{))}\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(5) $\mathcal{C}[\texttt{(let (v (car v}_1\texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v},\ \texttt{(fieldOf v}_1\texttt{ 1))},\ (\texttt{v}_1,\ \texttt{(cons v any))}\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(6) $\mathcal{C}[\texttt{(let (v (cdr v}_1\texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v},\ \texttt{(fieldOf v}_1\texttt{ 2))},\ (\texttt{v}_1,\ \texttt{(cons any v))}\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(7) $\mathcal{C}[\texttt{(let (v (f v}_1\texttt{ ... v}_n\texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

(a) If `f` is not bound, or `f` is currently being defined:

$::= \mathcal{C}[\mathit{expr},\ \mathrm{E}^{(n)}]$

(b) If `f` is locally bound to $\texttt{v}_f$ and $\texttt{v}_f$ is untyped:

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v}_1,\ \texttt{(fieldOf v}_f\texttt{ 1))} \ldots (\texttt{v}_n,\ \texttt{(fieldOf v}_f\texttt{ n))},$
$(\texttt{v},\ \texttt{(fieldOf v}_f\texttt{ n+1))}\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(c) If `f` is bound with type $\texttt{(->}\ \tau_1 \ldots \tau_n\texttt{)}$:

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v}_1\ \mathit{tau}_1) \ldots (\texttt{v}_n\ \mathit{tau}_n)\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(d) If `f` is bound with type $\texttt{(or (->}\ \tau_1 \ldots \tau_n\texttt{)} \ldots \texttt{(->}\ \tau_{m+1} \ldots \tau_{m+n}\texttt{))}$:

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v}_1,\ \mathit{tau}_1) \ldots (\texttt{v}_n,\ \mathit{tau}_n)\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\} \cup \ldots \cup$
$\{\mathrm{E}_i \cup \{(\texttt{v}_1,\ \mathit{tau}_{m+1}) \ldots (\texttt{v}_n,\ \mathit{tau}_{m+n})\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]$

(e) If `f` is a member of {`equal? eql? eq? =`}:

$::= \mathcal{C}[\mathit{expr},\ \{\mathrm{E}_i \cup \{(\texttt{v}_1,\ \texttt{(= v}_2\texttt{))},\ (\texttt{v}_2,\ \texttt{(= v}_1\texttt{))},\ (\texttt{v},\ \texttt{true})\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\} \cup$
$\{(\texttt{v},\ \texttt{\#f})\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}$

(8) $\mathcal{C}[\texttt{(let (v (if v}_{guard}\ \mathit{expr}_{true}\ \mathit{expr}_{false}\texttt{)) } \mathit{expr}\texttt{)},\ \mathrm{E}^{(n)}]$

$::= \mathcal{C}[\mathit{expr},\ \mathcal{C}[\mathit{expr}_{true}\ \{\mathrm{E}_i \cup \{(\texttt{v}_{guard},\ \texttt{true}),\ (\texttt{v},\ \textsc{finalvar}[\mathit{expr}_{true}])\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}] \cup$
$\mathcal{C}[\mathit{expr}_{false}\ \{\mathrm{E}_i \cup \{(\texttt{v}_{guard},\ \texttt{\#f}),\ (\texttt{v},\ \textsc{finalvar}[\mathit{expr}_{false}])\} \mid \mathrm{E}_i \in \mathrm{E}^{(n)}\}]]$

Figure 5.2.9: Constraint Generation

the type `string`, where the sub-definitions are named $+_{number}$ and $+_{string}$ respectively, the application of the type inference algorithm on the function

```
(define plus (lambda (x y) (+ x y)))
```

returns two types. The first type, `(-> number number number)` is generated using the assumption that the function `+` is instantiated with the sub-definition $+_{number}$ and the second type, `(-> string string string)`, with the sub-definition $+_{string}$.

As part of constraint generation, the name of any function currently being defined is kept in a list. The system also maintains a type environment with the names and types of all top level and primitive functions. Variables not in a position that indicates they are bound to a function are assigned a type of `any` unless they are bound in the environment in which case their type is the value they are bound to. Variables used as a function are assigned a type according to the following rules:

1. If the name is in the list of functions currently being defined, it is considered *recursive* and is given a type of `(-> any ... any)` with the appropriate arity.

2. If the name is bound using a `let`[4] construct, it is considered *local* and assigned a type of `(-> any ... any)` with the appropriate arity.

3. If the name is bound in the type environment, its type is the value bound to it in the environment. If the name is bound but not to a function, the entire expression being typed is given a value of `void`.

4. If none of the previous cases hold, the name is an unbound symbol and is given a type of `(-> any ... any)` with the appropriate arity.

Examining the constraint sets of the function `length`, found in figure 5.2.10, one can see that there are two constraint sets generated, one for each branch of the `if` statement that `length` contains. The constraining type `any`, associated with many of the variables, is a default constraint supplied by the system when no other constraints are discovered. At this point, it is possible to see that the first constraint set describes the "base case" of the function, terminating the recursion and having a type of `(-> null 0)`. The second set is more complex but is identifiable as describing the recursive case with a "generalized" type

[4] A `let` in the intermediate form is equivalent to a `letrec` in Scheme.

Constraint Set 1 ((`null? x`) $\Rightarrow$ `#t` indicating the "then" branch)

`length:`	`(-> x if-expr)`	`if-expr:`	`(= zero)`	`result:`	`any`
`x:`	`null`	`zero:`	`0`	`one:`	`any`
`bool:`	`#t`	`cdr-x:`	`any`	`sum:`	`any`

Constraint Set 2 ((`null? x`) $\Rightarrow$ `#f` indicating the "else" branch)

`length:`	`(-> x if-expr)`	`if-expr:`	`(= sum)`	`result:`	`(= if-expr), number`
`x:`	`(cons any cdr-x)`	`zero:`	`any`	`one:`	`1, number`
`bool:`	`#f`	`cdr-x:`	`any`	`sum:`	`number`

Figure 5.2.10: Constraints Generated for Length

of `(-> (cons any any) number)`. The generality of this type is a product of constraint generation not identifying the equivalence, implied by recursion, of the variables `cdr-x` and `x`.

This algorithm may potentially generate a number of constraint sets that is exponential to the number of control flow paths. To prevent this from occuring, a partial solution of the constraint sets is done during constraint generation to identify and discard constraint sets that indicate unreachable control flow paths. For example, when the guard of an `if` expression is the constant `#t`, no duplication of the constraint set occurs. This duplication is prevented since the intersection of the guards assumed value (`#t`) and its required value (`#f`), in the else clause, results in the value `void`, therefore indicating an unreachable control flow path. Experience has revealed that this implementation technique is effective in limiting the number of constraint sets generated.

5.2.3 Constraint Solution

Once all constraints have been generated, the final stage is to solve each set of constraints using an intersection based approach. Intersection can be used due to the simplistic nature of the constraints generated. Each constraint represents a local maximum on the set representing the type of a quantity, and the quantity must satisfy all constraints upon it, therefore the final value (type) of any variable can be determined by intersecting the types inferred by the constraints.

As part of the constraint solution phase, all the type constraints upon a variable must

$$\textsc{intersect}[\tau_1,\ \tau_2] \Rightarrow \tau$$

$\textsc{intersect}[\texttt{any},\ \tau] ::= \tau$
$\textsc{intersect}[\tau,\ \texttt{any}] ::= \tau$

$\textsc{intersect}[\texttt{true},\ \tau] ::= \tau \text{ if } \tau \neq \texttt{\#f}$
$\textsc{intersect}[\tau,\ \texttt{true}] ::= \tau \text{ if } \tau \neq \texttt{\#f}$

$\textsc{intersect}[\tau,\ \texttt{c}] ::= \texttt{c} \text{ if } \texttt{c} \in \tau$
$\textsc{intersect}[\texttt{c},\ \tau] ::= \texttt{c} \text{ if } \texttt{c} \in \tau$

$\textsc{intersect}[\tau_1,\ \tau_2] ::= \tau_1 \text{ if } \tau_1 = \tau_2$

$\textsc{intersect}[\texttt{(cons } \tau_1\ \tau_2\texttt{)}\ \texttt{(cons } \tau_3\ \tau_4\texttt{)}]$
$\quad ::= \texttt{(cons } \textsc{intersect}[\tau_1,\ \tau_3]\ \textsc{intersect}[\tau_2,\ \tau_4]\texttt{)}$

$\textsc{intersect}[\texttt{(->} \ \tau_1\ \ldots\ \tau_n\texttt{)}\ \texttt{(->}\ \tau_{n+1}\ \ldots\ \tau_{n+m}\texttt{)}]$
$\quad ::= \texttt{(->}\ \textsc{intersect}[\tau_1,\ \tau_{n+1}]\ \ldots\ \textsc{intersect}[\tau_n,\ \tau_{n+m}]\texttt{)} \text{ where } n = m$

$\textsc{intersect}[\tau_1,\ \tau_2] ::= \texttt{void} \text{ when no other case applies}$

Figure 5.2.11: Definition of Type Intersection

be intersected, which is done using the algorithm given in figure 5.2.11. It is important to note that this algorithm does not have a rule for the type constructor or. The or constructor is only used after type inference is complete to combine the type for each control flow path. Branches in control-flow can only occur in a function abstraction since Scheme does not provide any other mechanism to bind unevaluated code to a symbol; thus, all or types indicate overloaded functions. Examining $\mathcal{C}$ reveals that these types are removed by the algorithm and are never entered into the constraint environment. Therefore, intersection of or types is not needed since they never occur.

Constraints upon variables represent sets of values assigned to the variable at different points throughout the program. Since a variable's value is fixed for a function invocation, barring the occurrence of an assignment such as set! which will be ignored, the variable

Let the initial solution S_1 be: $\{(v_1, \text{any}) \ldots (v_j, \text{any})\}$

```
The notation: v, w describes variables
              N is an integer
              $S_i(v)$ refers to the value of v in $S_i$
              $S_i(v)[N]$ refers to the Nth component of $S_i(v)$
              $\text{QUANTIFY}_i[\gamma] = \gamma$ with all v replaced with $S_i(v)$

loop {
    for each $(v, \gamma)$ in C
        case $\gamma$ of (= w):           $S_{i+1}(v) \leftarrow S_i(v) \cap S_i(w)$
                  (fieldOf w N):  $S_{i+1}(v) \leftarrow S_i(v) \cap S_i(w)[N]$
                  default:        $S_{i+1}(v) \leftarrow S_i(v) \cap \text{QUANTIFY}_i[\gamma]$
    $i \leftarrow i + 1$
} until $S_{i-1} = S_i$
```

Figure 5.2.12: Constraint Solution Algorithm

must be a member of all the constraining sets simultaneously. Therefore, the sets can be intersected to find the smallest possible set of values and the most accurate type for the variable.

For any set of constraints, $C = \{(v_1, \gamma_1), (v_2, \gamma_2) \ldots (v_n, \gamma_n)\}$, over j variables where $1 \leq j \leq n$, the solution to the constraint set is described using the algorithm in figure 5.2.12.

Since the transference of type information among variables is also accomplished with constraints, it is insufficient to simply intersect the constraints upon each variable. Thus, the algorithm in figure 5.2.12 must also account for this transference. The QUANTIFY function handles extended types by replacing variables with their current value, while the other constraints are handled by the first two cases of the case structure. In general, each constraint is handled by taking the component of the type being described and intersecting it with the current value of the variable being solved.

An iterative approach is taken to solving the constraint set. If constraints are considered nodes, and identical variables are connected by arcs, a constraint set can be seen as a graph. Since a change in any one point in the graph can have an affect on other points more than one arc away and each iteration of the solution algorithm transfers a

Solution to Constraint Set 1: ((`null? x`) $\Rightarrow$ `#t` = "then" branch)

`length:`	`(-> null 0)`	`if-expr:`	`0`	`result:`	`any`
`x:`	`null`	`zero:`	`0`	`one:`	`any`
`bool:`	`#t`	`cdr-x:`	`any`	`sum:`	`any`

Solution to Constraint Set 2: ((`null? x`) $\Rightarrow$ `#f` = "else" branch)

`length:`	`(-> (cons any any) number)`	`cdr-x:`	`any`
`x:`	`(cons any any)`	`result:`	`number`
`bool:`	`#f`	`one:`	`1`
`if-expr:`	`number`	`sum:`	`number`
`zero:`	`any`		

Figure 5.2.13: Constraint Solutions for Length

change over one arc, an iterative approach is required to ensure changes eventually spread throughout the entire graph. The procedure terminates when the value of all the nodes of the graph stabilize.

Applying this procedure to the constraints generated for the function `length`, the solution in figure 5.2.13 is obtained. This figure presents the final type inferred for each variable on each control flow path. As suggested by the constraints generated, the final type of the function is:

```
(or (-> null 0) (-> (cons any any) number))
```

5.2.4 Termination Properties of the Algorithm

In the preprocessing stage, the syntax tree of the function, whose type is to be inferred, is traversed twice, once for renaming and standardization and once for A-normalization. Since the traversal treats function names (in applications) as leaves of the syntax tree, no recursion can occur to prevent termination. This same argument holds for the constraint generation phase, but this time it is the intermediate representation which is traversed.

It is only slightly more difficult to reason that termination occurs during solution of the constraint sets. The solution algorithm halts when all the type values for variables become constant. In the worst case, every single constraint could be applied to every possible variable before this occurs. The absence of any recursive constraints prevents infinite length types from being generated by repeated application of the intersection algorithm. Thus, for a constraint set of m variables and n total constraints, a solution is always be reached after m^n intersections, and the algorithm always terminates.

5.3 Discussion of the Algorithm

This is a conservative algorithm. All variables are initially assigned the type of `any`, which is subsequently restricted by the constraints. Should insufficient information exist to restrict the type to its *precise* value, a more general type is inferred. This relaxation does not create incorrect types, only overly general ones.

One restriction of this type inference algorithm is that it does not identify the internal structure of a list by *unwinding* the recursion. This restriction is deliberate since lists in Scheme may be heterogeneous. Thus, if an overloaded function is mapped over a list, a different sub-definition may be used for each element of the list. Dispatch, necessary to select the appropriate sub-definition for each element, makes knowledge of the internal structure of a list unimportant in this case.

Although functional in nature, Scheme also has imperative features. In particular, primitive functions whose names end with '`!`', such as `set!`, are imperative in nature since they perform assignment. As part of the initial implementation, functions with side-effects were disregarded with respect to type inference. It remains as later research to investigate the integration of non-functional operators into this algorithm.

When constraints are generated for an `if` expression, knowledge about the type that a variable cannot have is present since it is known that the guard component of a conditional failed (i.e., evaluated to `#f`) in the else clause. This information is lost since no *exclusion constraints*, such as `(not null)`, are used to identify constraints that describe the types a variable cannot be a member of. Overly general types occur, when some Scheme programs are analyzed, as a result of losing this information. In the example given in figure 5.3.1, the second branch of the `cond` expression is never evaluated since the condition of `y` being `null` needed to satisfy the `and` clause causes the execution of the first branch of the `cond` expression. Therefore, the type of `(-> null null null)` associated with the unreachable code adds the unreachable value `null` to the range.

The next example, figure 5.3.2, demonstrates the results of the type inference algorithm

```
(define merge (lambda (x y)
  (cond ((null? y) (cons x y))              ;; case a
        ((and (null? x) (null? y)) '())     ;; case b
        (else (merge (car x)                ;; case c
              (merge (cdr x) y))))))

Has types:
   1. (-> any null (cons any null))
   2. (-> null null null)
   3. (-> (cons any any) any (cons any any))
```

Figure 5.3.1: Inaccurate Type Generation

```
(define num-member? (lambda (ls)
  (cond ((null? ls) #f)                     ;; case a
        ((number? (car ls)) ls)             ;; case b
        (else (num-member? (cdr ls))))))    ;; case c

Has types:
   1. (-> null #f)
   2. (-> (cons number any) (cons number any))
   3. (-> (cons any any) any)
```

Figure 5.3.2: An Example of Type Inference

on a slightly more complex example. Due to the considerable quantity of data generated by the intermediate steps, they are not reproduced here. It should be noted that type 3 covers type 2, and so the latter is not reported as part of the type for `num-member?`.

Rather than merging all the type information for each control flow path as is done in conventional typing, the type system attempts to express the generic nature of Scheme functions. This simplifies the addition of overloading since it becomes a matter of adding new types to the set of existing types for a function. Multiple types for a function are represented using disjoint union and are presented using the keyword `or`. This approach was inspired by Kind and Friedrich's work on EuLisp [35].

The A-normalization algorithm described and implemented is not a complete implementation of the theory described by Flanagan et al. [22]. However, neither is the implementation included in the same paper and reproduced here. The difference between the full theory and the implemented version is the replacement of the rule for `if` expressions with:

$\mathcal{A}$[`(if` v E_t E_f`)`, name, body]

 ::= `(let (`ν `(if` v $\mathcal{A}$[E_t, name, body] $\mathcal{A}$[E_f, name, body]`))` ν`)`

Both transformation rules produce similar results apart from changing the ordering of some of the `let` expressions. This ordering has no effect on the generation of constraints and so is not significant with regard to this type inference algorithm.

5.4 Recursion

The algorithm, as described, makes no attempt to detect and report recursive types, and can therefore, be described as a single step of Kaplan-Ullman fixed-point iteration [34]. Types are differentiated using the outermost type constructor only, implying that lists and pairs (which are both constructed using `cons`) are indistinguishable.

Since the type system does not have any recursive types, the types generated by recursive functions can only be represented by infinite sets such as {`(cons any any)`, `(cons any (cons any any))`, ...}. The set contains a series described by an infinite combination of some type constructors. Multiple steps of fixed-point iteration would not converge on a solution in the case of recursive data types as each expansion of the recursive type would extend the set by adding a value with one more layer of type constructors. Instead of resorting to some ad-hoc approach, which prevents infinite types from being generated, overloading was added to the language based on the previously described type system and inference algorithm.

The infinite types, which this system generates when modified to add the constraints inferred by recursion, occur because the type system lacks contractiveness [16]. Contractiveness can be described as a fixed upper bound (top) for all types when types are described using lattice terminology. This problem is intuitively understandable by considering the infinite type:

```
(cons any (cons any (cons any ...)))
```

which can occur during constraint solution and preventing a stable value from being achieved. To add contractiveness to the type system, there needs to be some fixed length type that this infinite type contracts to.

I initially believed that effective dispatch could be performed using only the outermost type constructor. After some experimentation, this was found not to be the case. The problem is not the inference algorithm however, but rather lies in the expressiveness of the type system. For example, it is impossible to express a function requiring two equal length lists as parameters. This inadequacy lead to addition of the `lambda++` syntax to allow the user to specify the dispatch code for a sub-definition.

One could improve the type inference algorithm by extending it to allow some arbitrary number of iterations of the inference process, and thus, establish a fixed-length for a type expression; but, this still does not fully resolve the problem. Fixed length types simply capture a longer fragment of an infinite type. This solution is accomplished by adding a new rule to the constraint generation algorithm, which, in cases of recursive function calls, adds the known type values of the domain and range of the function as constraints upon the arguments and return value at the call site. During constraint solution, when the size of the resulting types is examined and found to have reached some predetermined size, the algorithm is halted and the best approximation found for the type is returned.

Adding a mechanism to express recursion to the type system is eventually necessary in order to accurately express recursive types, though it then becomes difficult to use these recursive types to generate dispatch code. The only provided predicate identifying any recursive type is `list?`; this predicate identifies any `null` terminated series of `cons` types where each successive `cons` is in the second (`cdr`) field of the previous `cons`. Other patterns of recursion, such as binary search trees, are only determinable by writing Scheme functions which completely traverse the data structure while checking to ensure that it satisfies the necessary restrictions implied by the recursive pattern.

Currently under investigation is the addition of two new constraint types: `(domainOf` *function position*`)` and `(rangeOf` *function*`)`. These new constraints would be used to extend the type system and allow it to express the type of a function in terms of its inputs. For example, the function `length` would have a types of: `(-> null 0)` and `(-> (cons any domain[1]) number)`. The type `domain[1]` indicates that the second field of the `cons` type can be instantiated by any allowable value for the first parameter of the function. Domain and range types can also be used as type variables. The function `equal?` can be given a (partial) type of `(-> any domain[1] #t)` to describe the equivalence of the second parameter to the first when the function returns a true value. This equivalence is normally expressed with a type variable that must be instantiated in both positions with the same type. It is likely that different syntax will be needed to be make explicit these two different uses of `domain`.

Chapter 6

Motivation and Application

6.1 Motivation

In [56], the following four criteria are advocated as necessary for a programming language to support the cause of programming-in-the-large:

Separate Authority Various program components and abstract data types are maintained by separate authorities, with the authority for one component unable to modify any other component.

Generality Abstractions (particularly functions) should be allowed to have as large a domain as possible to improve their incidence of reuse.

Generalizability Abstractions should permit their domains to be broadened without this having an effect on their behaviour over the existing domain.

Incrementality Similar to generalizability, it should be possible to increase the domain of an abstraction in an incremental manner.

Overloading is an effective tool that, when added to a programming language, improves the language's ability to satisfy these criteria.

In Scheme, since there are no modularization facilities, it is difficult to satisfy the notion of separate authority. However, overloading does assist by allowing techniques to be used that promote the idea of separate authorities. For example, a programmer can design an abstract data type and, using overloading, can extend existing functions maintained elsewhere so that their domain is increased to include the new data type. In doing this, the existing behaviour of the function is not altered with respect to the authority that controls it. This capability also supports a better separation of authority by allowing each abstract data type to be responsible for only a subdomain of operators and functions common to many data types. As well, it permits data types to have all their functions grouped together in a single file, establishing the idea of a self-contained abstraction under separate authority.

Generality occurs as a natural result of the dynamic type system. Functions can always be applied to any values that do not cause a run-time test to fail, and thus, have their generality restricted only to prevent errors from occuring. This property is inherent to Scheme and is unaffected by the addition of overloading to the language.

Overloading improves both the generalizability and incrementality of Scheme. Existing functions can be incrementally extended to apply to larger domains, thus generalizing their behaviour, with no effect on current behaviour. However, generalization is not guaranteed to occur since it is possible for an overload definition to redefine the behaviour of the function for a subset of its existing domain. Overloading, while improving generalizability, can therefore be detrimental to it if used inappropriately.

Programming-in-the-large is not the only application of Scheme that is improved. Rapid prototyping now becomes simpler and easier. Functions can be defined such that they only handle an initial subset of their domain needed to implement a basic prototype. These functions can then be incrementally extended as the prototype becomes enhanced to demonstrate more advanced features. The initial subset is far easier to program and debug since it is smaller and less complex. Extending it is easy using overloading and does not require the reimplementation of existing code.

Code reuse is also improved. Any function can be made more general by overloading the functions it uses internally. The function does not need to be parameterized over its internally used functions and so does not need any rewriting. For example, by extending the `equal?` function, the `member` function can be used with new data types without any changes being made to it. The existing function `member` is now usable in new situations and yet remain unchanged over its existing domain.

6.2 An Example

Consider that in some existing library there is the sorting function:

```
(define sort (lambda (list-to-sort)
  (if (null? list-to-sort) '()
      (...
        (if (< (car list-to-sort) (car result-list))
            ...)
        ...)))
```

Elsewhere, in a file containing the abstract data type `point`:

```
(define point? (lambda (item)
  (and (list? item) (= 3 (length item)) (eq? 'pt (car item))))

(define < (lambda++ (x y)
  (and (point? x) (point? y))
  (< (sqrt (+ (square (second x))
              (square (third x))))
     (sqrt (+ (square (second y))
              (square (third y)))))))

(define display (lambda++ (p) (point? p)
                  (display (second p))
                  (display ",")
                  (display (third p))))
```

Using overloading, the ordering test '$<$' has been extended to apply to the new data type. Thus, the existing `sort` function in the basic library can be used without requiring any modifications. Reuse could also have been accomplished by parameterizing `sort` over the ordering test, but only by modifying the original function, or writing a new function with a different name. Programmers are also relieved of the burden of managing a variety of functions with similar names that accomplish the same thing. Finally, and most importantly, additional parameterizing generates functions with long and difficult to manage parameter lists. The efficiency of the system may be severely affected by the need to pass such large numbers of closures around.

The function `display` demonstrates the extension of an existing primitive function to account for the new data type. Again, as was the case with `sort`, this behaviour is attainable in standard Scheme as the following code indicates.

```
(define old-display display)

(define display
  (if (point? p)
      (begin (display (second p))
             (display ",")
             (display (third p1)))
      (old-display p)))
```

However, the above technique forces the programmer to keep track of all the names for `display` every time it is redefined. The overloading support provided essentially does this for the programmer and decreases the chance of errors while reducing the workload.

6.3 Overriding from Overloading

Overloading, when implemented using generic functions, such that resolution occurs at run-time, is a suitable tool for simulating the overriding of a virtual function as occurs in object oriented programming.

When implementing objects in this simulation, each class is treated as an abstract data type, where all instances of the class are tagged with a list containing the name of the class and all its supertypes. The list is ordered so that the object's class comes first, followed by its parent class and so on until the final element in the list is the name `Object`, which is the superclass of every class. Functions in the abstract data type fulfill the role that methods do for objects. Overloaded functions can be written that examine this list and select the function definition matching the earliest occuring name in the list. This procedure is effectively the same as the run-time dispatch occurring in dynamically typed, objected oriented languages, such as Smalltalk [23].

```
;; ** Class Document **
;; Constructor
(define make-document
  (lambda (id-number) (list '(DOCUMENT) id-number)))

;; Predicate
(define document?
  (lambda (doc) (member 'DOCUMENT (first doc))))

;; Identify a document
(define identify
  (lambda (doc) (assert (document? doc) (second document))))

;; Order documents, based on id-number
(define precede?
  (lambda (doc1 doc2)
    (assert (and (document? doc1) (document? doc2))
            (< (second doc1) (second doc2)))))

;; ** Class Cheque (SubType of Document) **
;; Constructor
(define make-cheque
  (lambda (id-number date amount)
    (list '(CHEQUE DOCUMENT) id-number date amount)))

;; Predicate
(define cheque? (lambda (chq) (member 'CHEQUE (first chq))))

;; Order cheques, based on date
(define precede?
  (lambda (chq1 chq2)
    (assert (and (cheque? chq1) (cheque? chq2))
            (< (third chq1) (third chq2)))))
```

Figure 6.3.1: Simulating Overriding

The function `precede?` in figure 6.3.1 is overloaded to apply to both objects of class `document` and of class `cheque`. If an object is a subtype of `document`, the definition for the subtype (class `cheque`) is used, since it occurs before that for the supertype. If no definition exists for the subtype, as is the case with the function `identify`, the definition for the supertype applies, which is as it would be in an object oriented language like Smalltalk. This works effectively because the subtypes are defined after their supertypes, which is a necessary restriction in all object oriented languages.

6.4 The SmallScheme Library

In order to test the extension and to demonstrate the utility gained from the addition of overloading to Scheme, a significant portion of the standard Smalltalk 80 library [23] was implemented using DEFINE^{++}. This implementation demonstrates that in a dynamically typed language, overloading and overriding are different names for the same fundamental concept of using run-time information to determine the computation to perform. However, overloading is a more general mechanism since it allows a function to be defined for any set of data types without imposing a subtype relationship among the data types as overriding does.

Due to the size and complexity of the library, several classes and their associated subclasses were omitted from this implementation. The `process` class was not implemented since Scheme is not a concurrent language and does not provide the programmer with process control features.[1] The Smalltalk graphics and user interface classes were also omitted. First, as a result of their complexity, it was deemed that implementing these classes would be beyond the scope of this thesis. Secondly, the SmallScheme library is intended to be hardware and platform independent, a criteria not achievable for graphic and interface functions. Other classes such as `String` and `Number` were omitted since it was not beneficial to use objects to simulate the basic types of Scheme.

The hierarchy diagram of figure 6.4.1 details the classes provided by the SmallScheme library. Apart from the member functions of each class, there are some functions, of a general nature and not part of any class, used by the implementation of the member functions. A few of these functions are:

[1]It is documented [25] that they can be implemented (though somewhat inefficiently) using continuations.

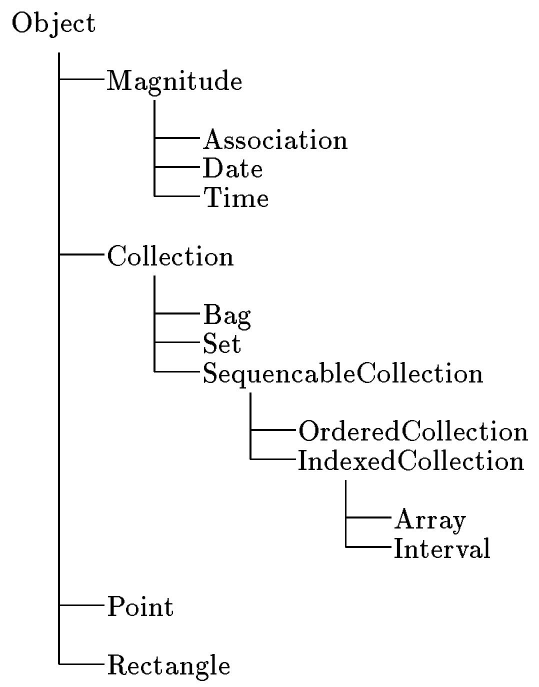

Figure 6.4.1: SmallScheme Class Hierarchy

- `last` which returns the last element of a list.
- `butlast` to remove the last element of a list.
- `foldl` to fold a function over a list from the left.
- `remove` to remove the first occurrence of an item from a list.
- `filter` to remove elements from a list based on a predicate.
- `apply-and` which "ands" the elements of a list.

The function `remove` is interesting in that, as well as operating on lists, it is also used to remove objects from instances of the class `Collection`, which is accomplished using overloading. The initial definition of `remove` is:

```
(define remove (lambda (x xs)
  (cond ((not (list? xs))
           (error "list needed for arg2 of remove"))
         ((null? xs)
           (error "item not in list"))
         ((equal? (car xs) x) (cdr xs))
         (else (cons (car xs) (remove x (cdr xs)))))))
```

Although it will be overloaded later, remove can be written as a conventional Scheme function. There is no requirement to use overloading to provide the initial behaviour for the function.

6.5 Objects

All "objects" in the SmallScheme library are members of class `Object`. In a functional language it is inconvenient to properly consider objects as structures to which messages are passed. Instead, the library implements objects in a functional manner, as a data structure which is passed into and returned by functions. Using this model, "methods" are the functions objects are passed to. While this is not a perfect implementation of the object-oriented paradigm, it is sufficient to demonstrate that the utility of overriding can be achieved using overloading.

All objects are lists, with the first element of each object being a "parentage tag" (p-tag). For example, a p-tag of the form `(set collection object)` indicates an instance of the class `Set` that is a subclass of `Collection`, which is in turn a subclass of `Object`. Any new fields added to the object by a subclass are appended to the end of the object. In keeping with the model provided by Smalltalk, only single inheritance is considered and implemented.

All objects, being subclasses of `Object`, can be passed to the following method functions:

- `(class` *object*`)`
  What is the class of *object*? Returns a symbol of lower case characters identifying the class of *object*. This symbol is the first element of the p-tag.
- `(class?` *class object*`)`
  Is *object* an instance of *class*? Returns either `#t` or `#f` depending on whether *class* is the first element of the p-tag.
- `(superclass?` *class object*`)`
  Is *object* a member of *class* or one of its subclasses? Returns the class of *object* if true or `#f` otherwise.
- `(subclass?` *class object*`)`
  Is *object* a member of *class* or one of its superclasses? Returns the class of *object* if true or `#f` otherwise.
- `(equal?` *object1 object2*`)`
  Is *object1* equal to *object2*? Returns either `#t` or `#f`.
- `(new` *class*`)`
  Return a new, uninitialized version of *class*.
- `(initialize` *initializers object*`)`
  Initialize *object* using *initializers* where *initializers* is a list of values appropriate for the class or a superclass of *object*. If the values are not appropriate, an error is returned.

Overloading is used by all other classes, since they are subclasses of `Object`, to perform any necessary specializations of these functions. For example, the function `equal?` is implemented in `Object` as:

```
(define equal? (lambda++ (x y)
                (and (subclass? 'object x)
                     (subclass? 'object y))
                #f))
(define equal? (lambda++ (x y)
                (and (class? 'object x)
                     (class? 'object y))
                #t))
```

To establish the default definition for object equality, the first definition, which is only used when the second or later definitions are not applicable, finds any two objects of the SmallScheme library to be unequal. This definition is immediately overridden so that any two objects that are both instances of class `Object`, which has no member variables, are found as equal.

The function `equal?` is overloaded for the class `Point` as:

```
(define equal? (lambda++ (x y)
                (and (class? 'point x) (class? 'point y))
                (and (equal? (x-coord x) (x-coord y))
                     (equal? (y-coord x) (y-coord y)))))
```

Any two points are equal if they are both points and they have the same x and y coordinates. Should one object be a `Point`, and the other an instance of some other class, then the first definition of `equal?`, which satisfies this condition, is the default condition established by `Object`.

6.6 The `Collection` Class

To better illustrate the implementation of the SmallScheme library, the `Collection` class is more closely examined. As do all other classes, `Collection` must provide any necessary overrides for member functions of `Object`. However, `Collection` is an abstract class, which is never intended to be instantiated. Its purpose is to provide a class to group the various subclasses of collection, namely: `Bag`, `Set` and `SequenceableCollection`. As such, `Collection` leaves the overriding of `Objects` member functions to these subclasses.

The class does however establish some default behaviour for the member functions of its subclasses using the following definitions:

```
(define empty? (lambda (x) #t))

(define member? (lambda (object collection) #f))

(define add (lambda (object collection)
  (error "can not add to class collection")))

(define remove (lambda++ (object collection)
  (class? 'collection collection)
  (error "can not remove from class collection")))

(define ssl:first (lambda (collection)
  (error "no objects in abstract class collection")))
```

The member `ssl:first`[2] is not part of the standard Smalltalk-80 library. It is a SmallScheme addition which, when overridden, provides a method for retrieving an arbitrary element from a collection. This function allows the class to implement `addAll` and `removeAll` in a manner that negates the need for them to be overridden. The functions `addAll` and `removeAll` for all collections are:

```
(define addAll (lambda (c1 c2)
  (if (empty? c1)
      c2
      (let ((item (ssl:first c1)))
        (addAll (remove item c1) (add item c2))))))

(define removeAll (lambda (c1 c2)
  (if (empty? c1)
      c2
      (let ((item (ssl:first c1)))
        (removeAll (remove item c1) (remove item c2))))))
```

[2]`ssl` is an abbreviation for SmallScheme Library.

The class Set describes collections of unique elements that are unordered and have no external "keys". A bag is similar to a set, except that multiple occurrences of its elements is possible. These two subclasses of Collection are now partially detailed and compared.

Both classes provide unique overrides for some of the member functions of class Object. The first function needed is new to generate new instances of each class. The overrides are:

```
(define++ new (lambda (class)
                (if (eq? class 'set)
                    '((set collection object) ()))))

(define++ new (lambda (class)
                (if (eq? class 'bag)
                    '((set collection object) ()))))
```

Before other members can be defined, each class needs to have an internal (private) structure imposed upon it. Both classes are implemented with lists, with the only difference being the allowance of multiple copies of the same element in the list used with Bag. Therefore, the basic functions defined in Collection are overridden as follows:

```
(define empty? (lambda++ (collection)
                  (or (class? 'bag collection)
                      (class? 'set collection)
                      (class? 'sequencableCollection collection)
                  (null? (cadr collection))))

(define member? (lambda++ (item collection)
                  (or (class? 'bag collection)
                      (class? 'set collection))
                  (if (member item (cadr collection)) #t #f)))

(define add (lambda++ (item bag)
              (class? 'bag bag)
              (list (car bag) (cons item (cadr bag)))))
```

```
(define add (lambda++ (item set)
              (class? 'set set)
              (if (member? item set)
                  set
                  (list (car set) (cons item (cadr set))))))

(define remove (lambda++ (item collection)
                 (or (class? 'bag collection)
                     (class? 'set collection))
                 (if (member? item collection)
                     (list (car collection)
                           (remove item (cadr collection)))
                     (error "object not in bag"))))

(define ssl:first (lambda++ (collection)
                    (or (class? 'bag collection)
                        (class? 'set collection))
                    (caadr collection)))
```

Some of the functions, such as `member?`, are implemented identically for more than one class, and so use the same override. This capability is easily accomplished by specifying, using `lambda++`, that the overload applies to both domains.

The two subclasses must also provide any necessary overrides for the member functions of class `Object`. For example, equality must be determinable between instances of `set`:

```
(define equal? (lambda++ (s1 s2)
                 (and (class? 'set s1) (class? 'set s2))
                 (and (= (length (cadr s1)) (length (cadr s2)))
                      (apply-and
                        (map (lambda (x) (member x s2)) s1)))))
```

To be equal, sets must be of the same length and have the same elements. Equality for class `Bag` is slightly more complex since there must also be the same number of occurrences of each element in each bag. Thus equality can be implemented using the member function `occurrences`:

```
(define occurrences (lambda (item bag)
  (cond ((not (class? 'bag bag))
            (error "occurrences not defined for class"))
        ((empty? bag) 0)
        (else (apply +
                (map (lambda (x)
                       (if (equal? item x) 1 0)) (cadr bag)))))))

(define equal? (lambda++ (b1 b2)
                 (and (class? 'bag b1) (class? 'bag b2))
                 (apply-and
                   (map = (map (lambda (x)
                                 (occurrences x b1)) b2)
                          (map (lambda (x)
                                 (occurrences x b2)) b2)))))
```

This concludes the analysis of the class `Collection` and its subclasses `Bag` and `Set`. It has illustrated the techniques used by the SmallScheme library to simulate overriding of virtual functions as it occurs in object oriented programming.

Chapter 7

Conclusion

7.1 Implementation Alternatives

As an alternative to implementing overloading using dynamic dispatch, it is possible to adopt a strategy whereby each sub-definition is tried until one executes without causing a run-time error. This alternative has the advantage of avoiding the limitations imposed by the primitives used for dispatch. Scheme, by using strict evaluation, prevents this approach from being adopted since the execution of a sub-definition may result in non-termination. Additionally, side-effects caused by the non-functional components of the language are not easily reversible adding complication to the approach. Also, when there are many sub-definitions with disjoint types to consider, this strategy becomes inefficient.

A more simplistic implementation of overloading can be obtained through the omission of the `define++` and `assert` syntax, which has the advantage of avoiding the need for any form of type inference. This approach was not adopted since it was the intention of the author to investigate the effectiveness of inference based dispatch generation.

The preprocessing stage of the inference algorithm generates a simplified intermediate representation of a Scheme program, not executable Scheme code. It would not be difficult to convert this intermediate form to Scheme since only two changes are needed. First, an extra set of parentheses must be inserted around the binding clause of `let` expressions and second, all occurrences of `let` need to be replaced with `letrec`. These changes would be useful if any form of optimization was to be performed on the intermediate representation.

Many existing type inference algorithms were not appropriate for use in this incremental approach to overloading, since functions are typed as they are defined, and therefore have undefined variable references. An alternate approach, where definitions are collected until a closed system is obtained, would permit the application of one of these existing algorithms.

7.2 Future Work

While successful in the adding of overloading to Scheme, this thesis has not fully investigated the issue. The work done so far has brought to light several interesting issues for future examination. Many decisions that were made during implementation, in order to simplify development, improve clarity or increase flexibility, are no longer valid and only reduce overall efficiency. Thus, not only should the work of this thesis be advanced, it should also be re-examined and improved with regard to efficiency issues.

The current implementation limits the user by only allowing the overloading of top-level functions, defined using the special form `define`. One open issue is the addition of localized overloading provided by the localized binding of sub-definitions in a `letrec`. Some initial work in this area, limited only by available time, indicates that the current constraint based approach is suitable for the addition of local overloading.

The strategy of having the most recent sub-definition occur first in the composite definition forces programmers to provide the default behaviour in the first sub-definition. During development of the SmallScheme library, it became apparent that, although not necessary, it would be useful to provide a mechanism for allowing one to select the location where the dispatch code for a sub-definition is placed in the composite definition. This capability would entail some new syntax, such as `define-default` which adds the new sub-definition to the end of the composite definition as opposed to the beginning. After additional experience with the system, it should become clear if this is needed.

As part of the generation of dispatch code, some simple optimizations are performed; however there is opportunity for more work in this area. The heuristics used to optimize pattern matching in ML appear to be potentially usable and should be explored for their applicability in this situation. If these heuristics are inappropriate, new heuristics should be developed to reduce the overhead of dispatch and permit faster execution times. Optimization is best done heuristically, as opposed to algorithmically, since it has been shown that complete optimization of pattern matching is an NP complete problem [10].

Future versions of Scheme, with overloading built into the implementation, may wish to provide additional primitives to perform more accurate dispatch. The main need is for a primitive of the form: `(of-type?` *type object*`)` which returns a value of `#t` if *object* has a type of *type*. This predicate is primarily needed to allow overloading based on the type of a functional argument, but, by being very general in nature, allows very simple dispatch code to be written.

Static overload resolution is another area that needs to be investigated. As part of the implementation, sub-definitions were renamed and maintained as individual functions, as opposed to their code being inserted into the composite definition, so that static overload resolution would be possible. If it can be determined, at definition time, that an overloaded function always resolves to call the same sub-definition, then the renamed sub-definition should be inserted in place of the overloaded function. As would be the case with the optimization of dispatch code, static overload resolution should lead to improvements in execution time.

The system is also limited by the effectiveness of the type inference algorithm. The algorithm, as it is currently implemented, does not account for the side-effects caused by imperative style assignment functions in Scheme, such as `set!`. The function `call--with-current-continuation` is another Scheme function which that is traditionally problematic for type inference. To provide better types, it would be desirable to improve the inference algorithm to account for the effects of these functions. It may also be worthwhile to replace the existing inference algorithm with one of the others mentioned in section 2.4.

The adaptation of Gordon Vreugdenhil's framework for on-line partial evaluation [51] was considered as an alternative type inference mechanism. From personal communication it was determined that the approach was feasible and would have overcome many of the limitations of the simple algorithm implemented. Time constraints eventually ruled out this approach leaving it as an area for future investigation.

Since the current specification for Scheme [14] considers macros as an implementation option and only details them in an appendix, every language implementation provides its own version of macros. As a result, the extension described in this thesis has only been implemented for the SCM (Version 4E1) dialect of Scheme. There were two significant reasons for this choice. First and foremost, SCM has been ported to almost every commonly used architecture, where it executes in a very small memory space. Secondly, SCM provides macros which are identical to those used in Common Lisp and are well documented and easy to use. Once the previously mentioned issues in this section are examined, it would be beneficial to have this extension rewritten using the macro syntax of some of the other major Scheme implementations.

As of the writing of this thesis, selected members of the Scheme community are meeting to determine the future shape of the language. It is anticipated that results of these meetings will be published as the *Revised^5 Report on the Algorithmic Language Scheme.* Personal communication with Matthias Felleisen, who has been participating in these meetings, has indicated that the resulting report will introduce standards for the syntax and semantics of macros in Scheme. When this occurs, a major revision of the extension should be performed and the extension recoded using the new macro syntax.

7.3 Summary

This thesis has described a simple extension to Scheme that improves the language's capability for code reuse with only a minimal amount of new syntax. The extension attempts to preserve the existing semantics of Scheme so that, for the most part, existing code runs without change. The use of macros to provide the new features allows overloading to be easily added to any implementation of Scheme.

The most significant benefit imparted is an improved ability to define abstract data types that reuse existing Scheme code. While the advantages provided by the extension can be simulated in standard Scheme, doing so is very cumbersome to the programmer. The end result is an increased ability to modularize programs and to express incremental abstract data types.

Although it is possible to generate dispatch code based on inferred types, it is difficult to do so efficiently and accurately. When a simple type system is used where only the outermost type constructor is inferred, types can be inferred quickly but at the cost of effective dispatch. To generate better dispatch code requires a rich type system, which in turn complicates the inference of types. The simple inference algorithm implemented here has been revealed to lack the richness necessary for effective overloading based solely upon inferred types.

In summary, by allowing the programmer to provide any necessary dispatch code, it is possible to extend Scheme with a relatively transparent implementation of overloading. Collecting a distributed set of sub-definitions and then merging them into a generic style function is an effective and simple method of accomplishing this task.

Bibliography

[1] Adams, N., and Rees, J. Object-oriented programming in Scheme. In *Proceedings of the Conference on Lisp and Functional Programming* (1988), Association for Computing Machinery, pp. 277–288.

[2] Aho, A., Sethi, R., and Ullman, J. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[3] Aiken, A., and Murphy, B. Static type inference in a dynamically typed language. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages* (January 1991), Association for Computing Machinery, pp. 279–290.

[4] Aiken, A., and Wimmers, E. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming and Computer Architecture* (1993), Association for Computing Machinery, pp. 31–41.

[5] Aiken, A., Wimmers, E., and Lakshman, T. Soft typing with conditional types. In *Proceedings of the Twenty-first ACM Symposium on Principles of Programming Languages* (January 1994), Association for Computing Machinery, pp. 163–173.

[6] American National Standards Institute. *FORTRAN Programming Language Standard*, 1978. Standard X3.9-1978 (FORTRAN 77).

[7] Appel, A., and Jim, T. Continuation-passing, closure-passing style. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages* (January 1989), Association for Computing Machinery, pp. 293–302.

[8] Backus, J., et al. FL language manual, parts 1 and 2. Tech. Rep. RJ 7100, International Business Machines, 1989.

[9] Baker, H. The Nimble type inferencer for Common Lisp-84. Unpublished report, December 1991.

[10] Baudinet, M., and MacQueen, D. Tree pattern matching for ML. (Extended Abstract) Unpublished, December 1985.

[11] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report. Tech. Rep. 31, DEC Systems Research Center, Palo Alto, CA, 1988.

[12] Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4 (1985), 471–522.

[13] Cartwright, R., and Fagan, M. Soft typing. In *Proceedings of the Conference on Programming Language Design and Implementation* (1991), Association for Computing Machinery, pp. 278–292.

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

[15] Cormack, G., and Wright, A. Type-dependent parameter inference. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 1990), Association for Computing Machinery, pp. 127–136.

[16] Courcelle, B. Fundamental properties of infinite trees. *Theoretical Computer Science 25* (1983), 95–169.

[17] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[18] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. Tech. Rep. CIS-91-2, Department of Computing and Information Sciences, Kansas State University, 1991.

[19] Duggan, D., Cormack, G., and Ophel, J. Incremental overloading for software reuse (draft). Unpublished, 1992.

[20] Duggan, D., Cormack, G., and Ophel, J. Kinded type inference for parametric overloading. *Acta Informatica 33* (1996), 21–68.

[21] Flanagan, C., and Felleisen, M. Set-based analysis for full Scheme and its use in soft-typing. Tech. Rep. TR95-253, Department of Computer Science, Rice University, October 1995.

[22] Flanagan, C., Sabry, A., Duba, B., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the Conference on Programming Language Design and Implementation* (1993), Association for Computing Machinery, pp. 237–247.

[23] Goldberg, A., and Robson, D. *SMALLTALK-80: The Language.* Addison-Wesley, Reading, Massachusetts, 1989.

[24] Gosling, B., Joy, B., and Steele, G. *The Java$_{TM}$ Language Specification.* Addison-Wesley, Reading, Massachusetts, 1996.

[25] Haynes, C., Friedman, D., and Wand, M. Continuations and coroutines. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1984), Association for Computing Machinery, pp. 293–298.

[26] Heintze, N. Set based analysis of ML programs. Tech. Rep. CMU-CS-93-193, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, July 1993.

[27] Henglein, F. Dynamic typing. In *Proceedings of the European Symposium on Programming* (1992), vol. 582 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 233–253.

[28] Henglein, F., and Rehof, J. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proceedings of the Conference on Functional Programming and Computer Architecture* (1995), Association for Computing Machinery, pp. 192–203.

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

[30] Hudak, P., Jones, S. P., Wadler, P., et al. Report on the programming language Haskell, a non-strict purely functional language. *ACM SIGPLAN Notices 27*, 5 (May 1992).

[31] Jensen, K., and Wirth, N. *Pascal: User Manual and Report*, second ed. Springer-Verlag, New York, 1974.

[32] JONES, M., AND MUCHNICK, S. Binding time optimization in programming languages. In *Proceedings of the Third ACM Symposium on Principles of Programming Languages* (January 1976), Association for Computing Machinery, pp. 77–94.

[33] KAES, S. Parametric overloading in polymorphic programming languages. In *Second European Symposium on Programming* (March 1988), H. Ganzinger, Ed., vol. 300 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 131–144.

[34] KAPLAN, M., AND ULLMAN, J. A scheme for the automatic type inference of variable types. *Communications of the ACM 27*, 1 (1980), 128–145.

[35] KIND, A., AND FRIEDRICH, H. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation 6*, 1/2 (August 1993), 159–175.

[36] MA, K.-L., AND KESSLER, R. TICL – a type inference system for common lisp. *Software – Practice and Experience 20*, 6 (June 1990), 593–623.

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

[38] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[39] QUEINNEC, C. Meroon: A small, efficient and enhanced object system. Tech. Rep. LIX.RR.92.14, INRIA-Ecole Polytechnique, 1992.

[40] RÈMY, D. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages* (January 1989), Association for Computing Machinery, pp. 77–87.

[41] REYNOLDS, J. Automatic computation of data set definitions. *Information Processing 68* (1969), 456–461.

[42] ROBINSON, J. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery 12*, 1 (January 1965), 23–41.

[43] ROUAIX, F. Safe run-time overloading. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages* (January 1990), Association for Computing Machinery, pp. 355–366.

[44] Sabry, A., and Wadler, P. A reflection on call-by-value. In *Proceedings of the International Conference on Functional Programming* (1996), Association for Computing Machinery, pp. 13–24.

[45] Shivers, O. Data-flow analysis and type recovery in Scheme. In *Topics in Advanced Language Implementation*, P. Lee, Ed. MIT Press, 1991, ch. 3, pp. 47–88.

[46] Smith, G. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, Ithaca, NY, August 1991.

[47] Smith, G. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming 23* (1994), 197–226.

[48] Smith, G., and Volpano, D. On the complexity of ML typability with overloading. In *Proceedings of the Fifth ACM Functional Programming Languages and Computer Architecture Conference* (August 1991), J. Hughs, Ed., vol. 523 of *Lecture Notes in Computer Science*, Association for Computing Machinery, Springer-Verlag, pp. 15–28.

[49] Stroustrup, B. *The C++ Programming Language*, second ed. Addison-Wesley, Reading, Massachusetts, 1991.

[50] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ansi/mil-std-1815a-1983 ed., February 1983.

[51] Vreugdenhil, G. *A Framework for On-line Partial Evaluation.* PhD thesis, University of Waterloo, 1996.

[52] Wadler, P., and Blott, S. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages* (January 1989), Association for Computing Machinery, pp. 60–76.

[53] Wang, E., and Hilfinger, P. Analysis of recursive types in Lisp-like languages. In *Proceedings of the Conference on Lisp and Functional Programming* (1992), Association for Computing Machinery, pp. 216–225.

[54] Wegbreit, B. The treatment of data types in EL1. *Communications of the ACM 17*, 5 (1974), 251–264.

[55] Wright, A. *Practical Soft-typing for Scheme.* PhD thesis, Rice University, 1994.

[56] Wright, A., and Cartwright, R. A practical soft type system for Scheme. In *Proceedings of the Conference on Lisp and Symbolic Computation* (1994), Association for Computing Machinery, pp. 250–262.

[57] Wright, A., and Duba, B. *Pattern Matching for Scheme, Version 1.12*, May 1995. Available at World Wide Web URL `ftp://cs.rice.edu/public/wright` as `match.ps.Z`.

[58] Wulf, W. A., Russell, D. B., and Habermann, A. N. BLISS: A language for systems programming. *Communications of the ACM 14*, 12 (December 1971), 780–790.