1990

Data Constraints in Function-Oriented Languages

Earl Stavis Harris

College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

DATA CONSTRAINTS

IN

FUNCTION-ORIENTED LANGUAGES

A Thesis

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

Master of Science

---

b y

Earl Stavis Harris Jr.

1990

APPROVAL SHEET

This thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(signature)
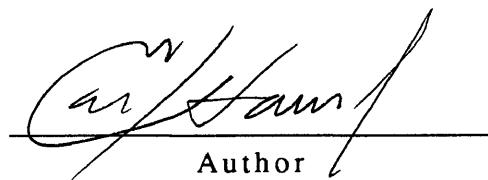
Author

Approved, May 1990

(signature)
Larry Morell

(signature)
Robert Noonan

(signature)
Robert Collins

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

ABSTRACT

This study explores the boundaries of a data constraint method for developing software. The technique allows programmers to impose constraints on data objects. The system works best when applied to function-oriented languages. (In a function-oriented language, subprograms model mathematical functions. The use of erasable memory cells are discouraged.)

Data constraints can heighten a person's understanding of specific objects by discriminating objects within their class. However, programmers who use data constraints in a conventional programming language only get a limited understanding of their software products. Therefore, programmers should use a language that is simple to understand, yet suitable for practical application development. For example, to make Scheme support data constraints, this thesis proposes adding extensions to the language. These extensions are based on a new function declaration which can describe its input. This thesis includes a formal definition for this enhancement which refers to Scheme's formal semantics.

This report presents a study that shows the advantages of using the extended Scheme to build a software system. In contrast to the positive aspects of the study, the chief drawback is that the system checks all data constraints as the program executes. This practice degrades performance. The feasibility of building a system which can validate most of the data constraints before the program executes is one of the open questions left by the study.

DATA CONSTRAINTS
IN
FUNCTION-ORIENTED LANGUAGES

1. The Data Constraint Thesis

In software engineering, imposing constraints on program data objects is a powerful technique with vast potential. But today, its potential is unrealized because programmers use the technique improperly. This leads us to an important thesis:

A powerful software design method, which allows programmers to impose useful constraints on data objects, is best suited for simple, orthogonal (flexible) function-oriented languages. (A function-oriented language supports the applicative style of programming, like a functional language. Unlike a functional language, a function-oriented language includes the assignment statement. The assignment statement is an impurity in applicative programming.)

To support this thesis, this report will show the following.

1. The data constraint method is a useful technique in other academic disciplines. So applying this technique to computer science should have many potential benefits.

2. Current programming systems cannot support this important method adequately. For example, a typing system alone does not exploit the full power of data constraints.

3. By using a language that can support the method adequately, the programmer can document and check more data constraints. Tools can check more constraints automatically. Scheme is a language that can support this method, because it is function-oriented.

4. Given the simple data constraint system described in point 3, one can build a system that will provide extended constraint checking support.

5. Researchers should study the data constraint method further.

2. How This Thesis is Organized

Section three of this thesis serves as an introduction to the data constraint method. It describes the design method and explains its use in other disciplines. It also gives applications of the data constraint method in computer science.

Sections four, five, and six of this thesis provide motivation for a new data constraining technique. Section four of this thesis explains why present applications of the data constraint method in computer science are unsatisfying. Section five shows why software engineers should be motivated to push the data constraint method to a higher plane. Section six explains why current data constraint methods are unable to realize their true potential.

Section seven reviews this thesis' strategy for overcoming the obstacles of the data constraint method. This section will discuss Scheme in some detail. Those unfamiliar with this programming language should read the introduction to Scheme in the appendix.

The eighth section defines the theory that extends Scheme to support the data constraint method. It also discusses some interpretations of the theory that supports run-time data constraint checking.

The ninth section is a case study that uses the method to write a Scheme program. Software tools will provide run-time checking support. Here, the method allows the programmer to use data constraints in an effective way.

The tenth section, describes a Scheme tool that could be the basis for a system that provides compile-time checking support.

The final section summarizes what questions researchers must explore.

The appendix contains an introduction to the programming language Scheme. It is for those who are unfamiliar with this programming language.

3. What Data Constraints Are About

This section defines the term data constraint and gives many examples. These examples show that data constraints are important tools in many academic disciplines.

Technical writers define a noun by first classifying it as a member of a particular category. Then, they constrain the noun for that classification. For example, to classify something as a four-wheeled vehicle, one can get an image of a car, a truck, or a van. But by adding that this vehicle has an elegantly dressed driver and is pulled by horses around Central Park, one realizes that this is a description of a buggy. The explanation of how this four-wheeled vehicle differs from other four-wheeled vehicles is a constraint on the definition. Focusing our attention on the constraint increases our understanding of the object.

Also, set theoreticians might describe even numbers as the set of numbers such that any arbitrary member of that set, called E, must satisfy the condition that $E \bmod 2$ is equal to 0 (i.e. 2 divides E evenly). Here, "numbers" is the class of objects. The object schema E represents any arbitrary number which is an alleged member of the set of even numbers. The condition $E \bmod 2 = 0$ represents a characteristic function which describes the set of even numbers. It is also a constraint; this condition discriminates even numbers from non-even numbers. This set theoretical definition corresponds to our intuitive model of even numbers. The following figure describes the set.

```
┌────────────────────────────────────────────────┐
│  Numbers:  (the  universe)                       │
│                                                  │
│                      ⟋⎺⎺⎺⎺⎺⎺⎺⎺⎺⟍                 │
│                    ⟋              ⟍              │
│                   |                |             │
│  E mod 2 ≠ 0      |  E mod 2 = 0   |             │
│                   |                |             │
│                    ⟍              ⟋              │
│                      ⟍_____⟋                 │
│                                                  │
└────────────────────────────────────────────────┘
```
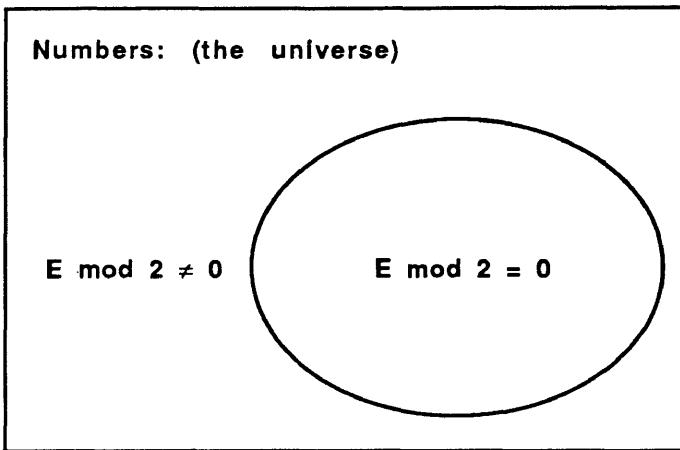
Like other academic disciplines, the understanding of data constraints is essential to programming.

When computer scientists design abstract, amorphous objects, data constraints can give these objects form and structure.

Also, throughout a program's life cycle, software engineers constantly check to see if their creations meet some informal condition. When they ask, "Does the program do what our customers expect?", they are checking constraints informally.

Furthermore, developers check constraints during debugging. A debug statement may display the value of some variable, so the developer can see if that variable satisfies a needed condition. For example, a debug statement may help a developer check if an index is still within the bounds of an array.

The preceding examples are cases where developers check informal constraints. The following are some examples of formal data constraints in programming.

Strongly typed systems allow developers to constrain variables to certain object classes in a formal way. This aids in detecting certain inconsistencies at compile-time (before the program executes). And these systems may provide features like subranges and subtypes to constrain these classes fur-

ther.

Another example of formal data constraints in programming is program verification. Program verification tries to prove that a program satisfies all data constraints, without executing the program. Since the 1960's, much research has been dedicated to this field. Researchers want to produce a medium in which the average programmer can use formal mathematics and logic to prove that their programs meet their specification. Some verification techniques define the properties of program constructs with a set of axioms (axiomatic semantics). Other techniques associate a mathematical model (denotational semantics) to a programming language's framework (the core language subset that defines the complete rules of the language).

Another example of formal program constraints is logic programming. By describing certain facts about a system, one develops a program implicitly. These facts are data constraints. Prolog is the main logic programming language; it is based on Horn clauses (a subset of logic that automata can execute efficiently). The following facts are from Clocksin & Mellish.[1]

```
member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).
```

These definitions describe the concept of list membership (when an element is a member of a list). The first definition says that if the given value and the head (first element) of the list are the same, then the given value is a member of the list. The second definition says that a given value is a member of the list, if that given value is a member of the tail of that list (the second through last element).

One can see that computer scientists use data constraints in various ways to help people understand systems.

---

[1] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, second edition, Springer-Verlag, Berlin, Germany, 1984, page 55.

4. The Data Constraint Method Has Unfulfilled Potential

This section will show how current usage of constraints in computer science is unsatisfying.

Most programmers specify constraints informally, using English prose within comments. There are problems with having comments as the only design documenting medium.

1. To some, comments are something to put into a program after it is finished, almost as an afterthought. Programmers take this view of comments because they are loosely coupled with the program. Comments are something placed within the program, but not part of the program itself. Furthermore, in a real programming environment, programmers reap greater rewards from producing more programs, even at the expense of less comments.
2. Comments can't affect a program, because they are not processed; the compiler ignores the information in the comments.

If documentation was part of the program medium and if it helped coders complete their programs, there would be more motivation to document programs. It would be advantageous if compilers could process some high-level design information and use it to confirm whether the program meets the requirements of the high-level design.

The program specification of customers are usually hazy, because they don't have a clear picture of what they want. To some extent, these type of specifications will always be hazy. But, if developers used more formal program specifications techniques when designing software, they could clarify certain aspects of a project. Furthermore, as one changes the program specification during the program's life cycle, one can better comprehend the consequences of the changes. And if the specification technique is formal enough, it might be possible to write tools that can detect which sub-programs are no longer consistent with the modified specification.

Though debug statements help programmers detect data constraint violations, data constraint described as debug statements do not exist throughout the life cycle of a program. Once the program is "debugged"; coders remove debug statements, because they destroy the program's aesthetics and degrade performance.

Though conventional strongly typed languages are helpful, there are many interesting constraints on objects which they cannot describe. For example, the correctness of a program may depend on the fact that a certain variable can only assume even values. But conventional strongly typed languages are not powerful enough to record this sort of constraint in a type definition. Given an enumerable type, a conventional type language might only have facilities to produce a subrange for that type. Here is such an example.

```
subtype INDEX is INTEGER range 1..100;
I : INDEX := 1;
```

Aside from its limited descriptive power, strongly typed, procedural languages have complicated rules for type agreement. Such rules are an obstacle to programmers. While the programmers want to solve a problem, they are distracted by the many peculiarities in the language medium they use.

Ada is an example of a procedural language that has introduced many complex rules to support strong typing to a high degree. In later sections, this thesis will discuss why such languages are complicated.

Though program verification sounds promising, for many reasons, formal program verification does not influence the average programmer. Program verification has become chiefly an intellectual, classroom exercise applied to simple program segments. The usage of program verification as a method to develop large scale, production quality programs is limited.

As one studies logic programs, one can see there are some constraints that they can't describe. In the list membership definition, the property that is conspicuously absent from the definition is

```
not member(X, []).
```

This states that it is not true that something is a member of an empty list (i.e. the empty list has no members). This is a reasonable thing to document. But negated axioms are not part of the syntax of Prolog (this prevents users from developing non-Horn clauses axioms). To simulate negated axiom, Prolog treats negation as failure (if Prolog can't prove it, then it is false). But the concept of "negation as failure" is not equivalent to logical negation. It would be advantageous if there were a logic programming language that allowed programmers to document negated axioms.

One can see that the application of data constraints in computer science has its problems.

5. The Data Constraint Method at Its Full Potential

In contrast to the formal and informal examples described earlier, the method described in this thesis includes

1. Formal, system-defined constraints.
2. Constraints that can record many interesting properties.
3. Constraints feasible to use in large scale, production quality programs.

What are the benefits of developing such a formal method? Our inability to develop formal definitions for software systems has prevented software engineers from making significant improvements in the software process. Consequently, software development lags behind hardware development. For example, Intel has recently development the 486 microprocessor. Yet, developers are still trying to perfect programs which exploit the features of the 286 microprocessor (which is two generations behind in hardware technology). So today, when computer scientists try to provide state-of-the-art technology to the world, perfecting the software has become the bottleneck. Since the bottleneck determines the flow of progress, scientists concentrate on removing obstructions at the bottleneck.
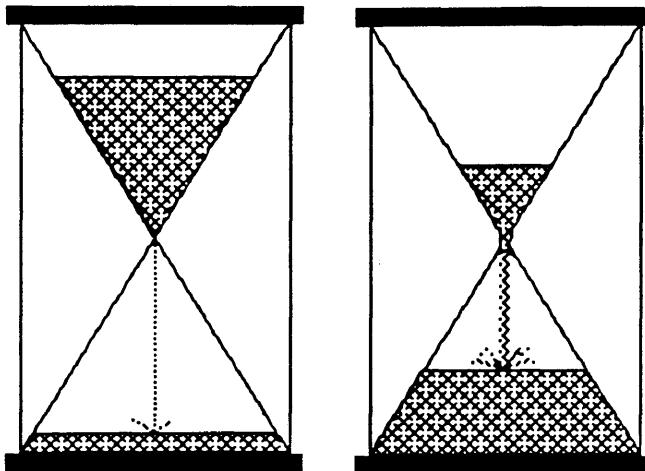
Today, as the cost of memory decreases and hardware becomes more efficient, customers request programs with more power. Likewise, the new software becomes more complex. As program specifications become more complex, the time necessary to perfect this software increases. Since software has become the bottleneck, computer scientists should concentrate on finding efficient ways to produce and maintain complex programs.

To support programming for large projects, modern day languages provide more powerful constructs. This is good; when a developer writes 100 lines of code today, that code segment will do more than the same amount of code in an older, less powerful language. But this does not solve the whole problem. To make it feasible for a developer to verify that the 100 lines of code meets its requirements, the developer must have complete understanding of the programming language. It is not enough for the developer to say, "it works, but I don't know why it works." If the complexity of the programming language is beyond human comprehension, then that language has not solved the problem. Any language that improves the software process must provide power within a simple model.

For the data constraint method to reach its potential, the high-level information about the program must be stored within the program. As a benefit,

translators can take advantage of this information to improve semantic analysis. If computer scientists put programs in the proper medium, a smart compiler will be able to manage much of the complexity that currently burdens the programmer. Furthermore, a smart compiler will be able to make many design decisions that programmers must make today. If software tools can manage more implementation details, programmers can manage more complex software projects. The programmers will work at a higher level. They will also devote their creativity to analyzing problems and producing elegant, easy to maintain, easy to reuse solutions. In effect, computer science would have removed the bottleneck.

One can compare the development of automatic heap management to the general state of programming today.

Before automatic heap management, the coder had to manage the heap directly. To ensure that the program met its requirements, the coder had to concentrate on avoiding dangling references (where the program has access to deallocated memory). He also had to write the program so it would avoid data loss (where all access to some allocated memory was lost, making collection of that data impossible). Preventing these situations was a distraction to the coder.

Now, when a programmer uses a language that has automatic heap management, he has no concern for dangling references or data loss. He concentrates more on the problem, and less on the medium used to solve the problem.

In the past, automatic heap management had large performance penalties. But today, there are clever ways to provide efficient automatic heap management. For a large class of applications, the overhead of heap management is acceptable. But a developer still cannot justify the cost of automatic heap management in some real time systems that have critical timing constraints.

The current state of programming is like a language which has no auto-

matic heap management. In both cases, programmers must manage many details. And future software analysis tools, like automatic heap management, should manage more programming details. The tools will catch more errors early. This reduces the concern for extensive testing. This will simplify the task of the programmer. Furthermore, future software tools will become efficient enough to help programmers in a large class applications. But there will probably be some applications where it will be necessary to manage all the details, to meet the application's requirements.

In summation, the goal of data constraints is to allow programmers to tell more about the program to the translator. Therefore, understanding how to apply formal data constraints to program development is one step to simplifying software development.

6. The Obstacles of Improved Software Analysis

A large part of problem solving involves defining the problem. This section will isolate why the data constraint method has not reached its true potential.

Since most strongly typed languages are also procedural, they have an intrinsic problem. It is difficult to introduce new technologies into a procedural languages, because they have inflexible frameworks. John Backus (Turing Lecture Award winner) argues that a language should have a small framework which can support a great variety of powerful features as changeable parts (the part of the programming language that the framework expects and the framework defines). Unfortunately, this is not characteristic of a procedural language. Since every detail of the computation changes state, the language designer must describe every detail of every feature into the state and its transition rules.[2] Hence, the framework must define most language features. The resulting framework becomes complex and rigid.

---

[2] John Backus, Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs*, *Programming Languages A Grand Tour*, Computer Science Press, page 178.

Because of the inflexible framework, the following cyclic scenario arises. Programmers yearn for a simple procedural language. Language designers produce such a language. But because of the inflexible framework and the scarcity of features, the programmers can't build what they want. The programmers then request more built-in features. Because of the inflexible framework, the complexity of the language framework increases greatly as new features are hooked into it. This augmentation process continues until the programmers say the system is too complicated. Then, the programmers yearn for a simple language again.

For example, consider the strongly typed procedural language Ada. It is based on Pascal. It was created because other languages in the late 70's were deficient in meeting the needs of the Department of Defense. Then, researchers produced supersets of Ada, called Ada-based design languages, to address the deficiencies of Ada. Meanwhile, many complain that the Ada programming language is too complex. So any superset of Ada would be even *more* complex.

An approach to breaking this cycle is to design a simple language with much power. To provide this, designers will need a simple, flexible framework. Then, they can add new features as changeable parts. No matter how complicated the features are, designers will build all new features on top of a simple, unifying concept.

Besides the limits of procedural languages, program verification and logic programming are limited because computer scientists have not found an effective way to use mathematics to analyze real programs.

Most program verification methods apply mathematics to programs by introducing a meta-language on top of the programming system. In functional correctness, a coder uses mathematical functions to document program input states and output states. These functions describe the program on a

higher level. In other methods, they use predicate calculus to describe program data spaces and the effect of statements on that state. These logical formulae describe the program on a higher layer.

Unfortunately, it is questionable whether adding a specification layer to a procedural language, which is different from the program medium, will make programming simpler. One of the reasons Von Neumann languages are complicated is that the assignment statement breaks the language into the world of expressions (values) and the world of statements (variables). It is counter-intuitive to expect to simplify the management of the complexity of this two-layered system by simply adding another, completely different layer. The complexity of the original language does not go away.

The language designer should ensure that programmers work in as few layers of development as possible. Therefore, an implementation language itself should allow developers to describe a program's data constraints. The complexity of a meta-system is an added burden.

Verification has difficulties in procedural programming languages because some program constructs are difficult to specify formally. Verification methods for procedural languages usually ignore such constructs. In effect, they only consider a subset of procedural languages. Usually researchers remove unstructured statements and shared references from the procedural language's working set.

Aside from this problem, it is debatable whether conventional machines can simulate the theoretical push-down automaton or Turing machines. While conventional machines have finite memory, both automata models have some form of countably infinite storage. Some claim that conventional machines are glorified forms of finite automata. But others argue that conventional computers are equivalent in power to a Turing machine. The latter group claims there is a way to simulate infinite memory. When the machine fills a

disk, the machine can suspended itself until a user gives the machine another disk. One assumes the user is always able to "buy" more disk space when the system "requests" it. However, it might be unreasonable for a machine to work this way; some machines must work independently.

Conventional machines exhibit their finiteness when they use limited or approximate representations of numbers. Though some programming data are called integers or reals, they do not behave like their mathematical counterparts. Computer integers are bounded; mathematical integers are not. Though there are finitely many computer reals between two points on a number line, there are infinitely many mathematical, rational numbers. It is improper to think that these program data types are equivalent to their mathematical counterpart. The following is a program used on the William & Mary Comprehensive Examination Question List (EQL).[3]

```
program Test;
var
      count : integer;
      tiny : real;
begin
      tiny := 1.0;
      count := 0;
      repeat
          tiny := tiny / 2.0;
          count := count + 1
      until (1.0 + tiny = 1.0);
      writeln(count)
end.
```

If count were a mathematical integer and tiny were a mathematical, rational number, then this program would never halt. The count would sim-

---

[3] Comprehensive Examination Question List, The College of William & Mary, 1989, page 4.

ply grow as tiny would approaches zero asymptotically. 1.0 + tiny would always be greater than 1.0, even if it was greater only by a very small value.

But count is not a mathematical integer and tiny is not a mathematical, rational number. When the floating point representation for 1.0 + tiny has more significant binary digits than the mantissa of a floating point register can store, rounding occurs. When count equals the length of the mantissa of the register used to store 1.0 + tiny, then the system rounds off the sum when it is stored. This makes the loop stop.

For a given system, the developer must decide either to make the software system simulate the unbounded, enumerable constructs of math or simulate conventional hardware. If one wants to simulate the unbounded, enumerable constructs of math, then the system should enter an error environment (or possibly request more memory), when the system cannot support the unbounded illusion. Likewise, when verifying a program the user must know which set of axioms to use, the axioms of numbers or the axioms for bounded computer integers and floating points.

In summation, software engineers are still searching for an effective way to analyze programs. The search is difficult for the following reasons.

- The fixed framework of most strongly typed procedural languages creates complexity problems.
- Adding layers of specification will not remove the complexity of the base language
- Unlike theoretical computer models, convention machines have difficulty representing unbounded, enumerable entities.

To overcome these problems, software engineers should consider alternate programming models with flexible frameworks (which allow introspection) and simple semantic definitions. Furthermore, all programmers should know how the behavior of bounded computer integers and reals deviate from the behavior of mathematical integers and rational numbers.

7. Overcoming the Obstacles

How can one get around the obstacles of improved software analysis?

Computer scientists cannot solve all the troubles of software development by adopting strongly typed, non-procedural languages. Though there are problems with procedural languages, there are also problems when one goes to the other extreme.

After John Backus' Turing lecture, many researchers developed functional languages (which avoid state machine semantics). Functional languages are characterized by their elimination of the assignment statement. Functional languages are conceptually simpler than procedural languages. In some programming problems, they are more powerful. But, there are some classes of problems difficult to describe and solve in a functional language, yet simple, elegant, and efficient to describe and solve in a language with assignment statements. For example, problem specifications that have some objects whose state is innately history-dependent may be unnatural to describe functionally. This is the case with specifications that have explicit parallelism, an interactive user interface, or input-output to secondary storage. Regrettably, functional languages are limited.

For general-purpose programming, one could use a language that supports applicative style programming, yet allows assignment as a necessary impurity. Scheme is such a language. This section will discuss why I chose Scheme to support data constraints.

In the revised report, Scheme provides essential decision procedures to determine the type of a value. For example, the procedures pair?, number?, and procedure? determine whether an object is a dotted pair, number, or procedure. These type checking functions are polymorphic and total. A program can apply any object in the Scheme domain to these functions.

It is natural for a Scheme programmer to use essential procedures to build new decision procedures for type checking. For example, a Scheme programmer can define variant types. To determine whether an object is either a string or an integer, one can use the following procedure.

```
( lambda (se)   ( or (string? se) (integer? se)))
```

Also, one can define a type with a constraint in Scheme. In section 3, even numbers are described as a class of integers with a constraint. Likewise, the predicate even? imposes a constraint on the predicate integer?.

```
( define even?
        (lambda (se) (and (integer? se) (zero? (modulo se  2
)))))
```

Unlike Scheme, strongly typed languages, like Ada, do not have provisions to support total, polymorphic functions. Therefore, strongly typed languages have limited type support.

Since strongly typed languages force users to declare a type for each object, these languages have difficultly creating objects that can be more than one type. If an Ada programmer wanted to create objects to store either static length strings (dynamic arrays are another more complicated issue) or integers, he would probably have to define a variant record. This is an inferior solution, because the user is burdened with the detail of managing a tag. A

programmer should only work with a tag, if it is part of the problem description. Otherwise, the tag is an unnatural and distracting implementation detail.

Though Scheme can support defining a type with an arbitrary constraint, like even?, Ada's subtype and derived type declarations are unable to produce such a characteristic function.

To correct this deficiency in a strongly typed language, the framework of the language must be extended to include an "any" type (which is compatible with all types). If Ada supplied a form of "any" type and basic type checking functions (like string? or float?), then programmers could write type checking procedures. (To some degree, generic formal parameters in Ada try to simulate what one can do with polymorphic functions.)

And Ada should include, a, simple, tagless, "oneof" type (which says the type of this object is one the following types). This addition is necessary because a "oneof" type is more appropriate than a variant record in certain situations.

But generally, polymorphic functions conflict with strong typing. Their introduction into a strongly typed computer language shows that the strong typing paradigm is sometimes too rigid. Furthermore, the introduction of polymorphic functions into a strongly typed language will undoubtedly increase the complexity of the language framework. For example, though "any" types seems like a good extension to Ada, it is possible that an "any" type would create complications in Ada's complex overloading rules.

Therefore, when compared to strongly typed procedural languages, the programming language Scheme has typing support with exceptional descriptive power. This provides power without sacrificing simplicity.

And, when compared to strongly typed procedural languages, Scheme has a framework with a simple, flexible semantic definition. Scheme's flexible framework can support extended data constraint facilities as changeable parts.

This makes the extensions easy to understand.

Unlike functional languages, Scheme's inclusion of an assignment statement makes it feasible for general purpose programming.

When one compares Scheme with Prolog, one can see that it is easier to describe logical conditions if one is not restricted to Horn clauses.

Also, since a Scheme program can be introspective, a Scheme coder can produce formal specifications and avoid the layered subsystem. And, programmers use Scheme to write large scale, production quality programs. Furthermore, Scheme has a mathematical, semantic definition. One can see that if more programmers adopt Scheme, they can reduce the gap between the mathematical computer theory and large scale, production quality programs.

8. Incorporating Data Constraints to Scheme

In the framework of Scheme, the lambda expression is the basic binding construct. Syntactically, the lambda expression is a parenthesized list whose first element is the name `lambda`. A parenthesized list of zero or more variables follows the name `lambda`. And one or more expressions follow this list. Here is a sample of the syntax in Backus-Naur Form.

```
<lambda expression> ::= ( lambda ( <variable>* )
<expression>+ )
```

The expression `(( lambda (x y) (- x y)) 3 4)` evaluates to -1; the `lambda` expression first binds the variables x and y to the values 3 and 4 and then subtracts y from x.

Scheme defines other more sophisticated binding forms as changeable parts. Some examples of these forms are the `let`, `let*`, and `letrec` expression. To build these extended forms, the `lambda` expression is a necessary component.

These extended binding forms are syntactically alike; all the extended forms discussed are parenthesized lists whose first element is their name (`let`, `let*`, or `letrec`). Unlike `lambda`, a parenthesized list of zero or more variable/value pairs follows the particular name. Finally, one or more expressions follow this list. For example, the `let` expression has the form.

```
<let expression> ::= ( let ( <variable/value pair>* )
      <expression>+ )
<variable/value pair> ::= ( <variable> <expression> )
```

This section adds new changeable parts to Scheme. To discriminate Scheme from Scheme with extensions, the extended Scheme will be called Scheme+ (pronounced "Scheme Plus"). Scheme+ is an abstract language extension; it defines many models. The basic form unique to Scheme+ is called the "constrained lambda" expression (or c-lambda, for short). Syntactically it is like the lambda expression, but there are two differences.

1. The first element in the parenthesis list is named c-lambda.
2. A special data constraint extension follows the argument list.

The following is the syntax of the constrained lambda.

```
<constrained lambda expression> ::=
      ( c-lambda ( <variable>* )
            <data constraint expression> <expression>+ )
```

Scheme+ extends the syntax of Scheme so all Scheme's key binding forms has a constrained version. In addition, this section defines a new form called a "constrained expression" (or c-expr, for short). It constrains the value of an expression. One can also use it to constrain the range of a given function.

The following example shows how to use constraints in Scheme+. Considering the following external definitions.

```
==>    ( define even?
            ( lambda (se)
```

```
                    (and (integer? se) (zero? (modulo se   2
))))
even?
==> ( define next-even (lambda (e) (+ e 2)))
next-even
==>
```

The code writer hopes that readers, by looking at the name and the body of the next-even definition, know that they should apply only even numbers to the procedure. The function next-even is meant to take an even number and produce the next even number. Unfortunately, the lambda expression is not descriptive enough to relay this fact; when given an odd number, the function is defined (it terminates normally). Considering the body of the lambda expression, one can only infer that the parameter e must be some number.

Maybe the coder should put in a comment. This comment should document the fact that this lambda expression is meant to take an even number. However, (next-even 5) will not be trapped as an error; it will produce 7. This is unfortunate because undetected errors can cause a program to have unexpected results. It is harder to find these types of errors, because the distance between the location of the error and the point where the error is detected could be large. If programs are harder to debug, then this will increases software development and maintenance costs.

Scheme+ will allow code developers to rewrite the definition this way.

```
==>   ( define even?
          ( lambda (se)
               (and (integer? se) (zero? (modulo se   2
)))))
even?
==> ( define next-even (c-lambda (e) (even? e) (+ e 2)))
next-even
```

$\Longrightarrow$

Now, after looking at the constrained lambda expression, readers can deduce that they should apply only even numbers to the procedure. A comment will be redundant. Various valid Scheme+ models should be able to detect (next-even 5) has a problem, by transforming (even? e) into a run-time constraint check.

Now we are ready to define the syntax more formally.

The Syntax of Scheme+

Definition: The *constrained version of a name* called x is the "c-" prefix followed by the name x.

Definition: *Scheme's key binding forms* are lambda, let, let*, and letrec.

Rule: The names c-lambda, c-let, c-let*, c-letrec, and c-expr are keywords in Scheme+. (The first four are the constrained versions of the names lambda, let, let*, and letrec.)

Rule: The following are called the constrained binding forms.

```
<constrained lambda expression> ::=
      ( c-lambda ( <variable>* )
            <data constraint expression> <expression>+
)

<constrained let expression> ::=
      ( c-let ( <variable/value pair>* )
            <data constraint expression> <expression>+
)
<constrained let* expression> ::=
      ( c-let* ( <variable/value pair>* )
            <data constraint expression> <expression>+
```

```
)

<constrained  letrec expression> ::=
        ( c-letrec ( <variable/value pair>* )
                <data constraint  expression> <expression>+
)

<constrained  expression> ::=
        ( c-expr <variable>
                <data constraint   expression> <expression>
)
```

Now that this section has defined the syntax of Scheme+, it will now give meaning to these syntactic forms.

The Semantics of Scheme+

Definition: an *unconstrained version of a constrained version name* n is n with the "c-" prefix removed. For example, the name lambda is the unconstrained version of the name c-lambda.

Definition: an *unconstrained version of a constrained binding form* f is f with its constrained version name replaced by its unconstrained version and its data constraint expression removed.

Definition: a *functional expression* is one that supports the applicative style because it does not treat variables as erasable storage cells. A functional expression cannot have side effects.

Comment: In the denotational semantics of Scheme, the function ε interprets syntactic forms in the Scheme language framework[4]. The function ε takes a

---

[4] Rees, Jonathan and William Clinger, ed. *Revised³ Report on the Algorithmic Language Scheme*, MIT Artificial Intelligence laboratory, September 1986, page 32.

syntactic form E, an environment U, and a continuation K (the future of a computation) and returns the result of evaluating E, given U and K. This result contains information about the return value of E and the updated version of the environment. This defines the language framework and allows one to describe certain properties.

Comment: In the following definitions, for any Scheme expression E, let E' be the result of taking E and reducing all its changeable parts to constructs in the framework.

Definition: A Scheme expression E is *defined* under arbitrary environment U and arbitrary continuation K, if the semantic function application ε (E', U, K) halts. Otherwise, E is *undefined*.

Definition: An expression E_1 is *at least as defined as* E_2 if and only if, when E_2 is defined, ε (E_1', U, K) = ε (E_2', U, K) for any environment U and any continuation K.

Comment: The following rules use the Scheme begin expression. This is a special form which is like the compound begin statement seen in other languages. Unlike the compound begin statement, the Scheme begin expression returns the value of the last expression.

Comment: In the following rules, let Ω be an undefined (non-terminating) Scheme expression.

Rule: Any arbitrary constrained lambda expression of the syntax

```
(c-lambda (<variable>*)
```

```
<data constraint expression> <expression>+)
```

is equivalent to some Scheme expression that is at least as defined as

```
(lambda (<variable>*)
     (if
          (and
               { <data constraint expression> is function-
al }
               <data constraint expression>)
          (begin <expression>+)
          Ω ))
```

Rule: Given any arbitrary constrained binding form with the syntax

```
({name} (<variable/value pair>*)
     <data constraint expression> <expression>+)
```

is equivalent to some Scheme expression that is at least as defined as

```
({name} (<variable/value pair>*)
     (if
          (and
               { <data constraint expression> is functional }
               <data constraint expression>)
          (begin <expression>+)
          Ω ))
```

The expression in curly brackets is a pseudo expression. Its determines whether the data constraint expression is a functional expression.

Rule: Any arbitrary constrained expression with the syntax

```
(c-expr <variable>* <data constraint expression> <ex-
```

pression>)

is equivalent to

```
((c-lambda (<variable>)
      <data constraint expression> <variable>) <expres-
sion>)
```

This theory defines the constrained expression as a changeable part; it is a special case of the constrained lambda.

This completes the theory of Scheme+. The previous schemata which represent the reduction of constrained binding forms are the minimal models (least defined). A constrained binding form is undefined if the data constraint expression is either not functional or not true. A Scheme+ interpretation can extend the domain of the minimal model in interesting ways. Given the minimal model, an interpretation can extend the domain by

1. Replacing the if expression's condition C with some condition C' such that C implies C'. As a benefit, the new model's *then* part accepts a superset of the domain accepted by the minimal model's *then* part.
2. Replacing Ω with a defined expression.

Also, this theory has an important property.

Theorem: Given an arbitrary constrained binding form, if the data constraint expression is functional and true, then the constrained binding form is equivalent to its corresponding unconstrained binding form, *under any interpretation of Scheme+.*

Proof: Consider any arbitrary constrained binding form whose the data constraint expression is functional and true. One can reduce the body of the corresponding minimal model so it is equal to the body of the corresponding un-

constrained binding form.

The following is the body of a minimal model.

```
(if
    (and
        { <data constraint expression> is functional }
        <data constraint expression>)
    (begin <expression>+)
    Ω )
```

Based on the given, both conjuncts are true; we can substitute them with true.

```
(if (and true true) (begin <expression>+) Ω )
```

These constant expression can be folded (simplified) as follows.

```
(if true (begin <expression>+) Ω )     ; conjunction is
true
    (begin <expression>+)                      ; then part always
taken
    <expression>+
```

The last reduction is valid, because the expression is within the context of a binding form, where the syntax allows one or more expressions. This shows that the minimal and unconstrained versions are equivalent when the condition is functional and true. Since this equivalence is a property of the minimal model, any model that is at least as defined as the minimal model also has this property (because they are equivalent where the minimal model is defined).

One may notice that the Scheme+ theory has no provision to constrain the state transitions of erasable storage cells. Scheme down-plays the state-

machine model; as a function-oriented language, Scheme encourage programmers to use an applicative style. Hence, this thesis chooses not to create "special case" constraints for erasable storage variables in this early stage of research. The constrained lambda and the constrained expression, constrains *values.* In the following example, notice how the constrained lambda creates the recursive analog to the loop invariant.

```
==>    ( define fact
             ( c-lambda (n)
                  ( natural? n )
                  ( if ( zero? n ) 1
                       (* n ( fact ( sub1 n )))))))
fact
==>
```

Notice that the constraint is on the input value, not the variable's erasable storage cell. There is nothing to prevent someone from assigning a value to n that is not a natural. Also, this thesis provides no "special case" analog to the statement constraint. Again, this thesis justifies this type of support, because the Scheme language encourages programmers to use the applicative style.

However, if a programmer wants to assign a new value to a variable, one can get benefits like an object constraint by assigning a constrained expression to that variable. Also, a sequence of "statement-like" expressions can have constrained lambdas (with no parameters) and constrained expressions interspersed to check state transitions.

Support for the object constraints should be reconsidered in a later stage of research. Fortunately, this type of support is not crucial when one stays close to the applicative style.

What is a valid Scheme+ interpretation? (At the point, we are not con-

cerned with implementing Scheme+ on a particular Scheme dialect. Rather we are describing the behavior of various versions of Scheme+.) A Scheme+ model candidate is a method that reduces all constrained binding forms into Scheme (like a changeable part) in a way that satisfies the theory.

For example, to support run-time constraint checking, we can take the minimal model and convert its body into the following.

```
(if <data constraint expression>
    (begin <expression>+)
    { enter error environment } )
```

This reduction is part of a binding form that is at least as defined as the minimal model. It simply replaces the if condition with its second conjunct and replaces Ω.

If the constraint is false when the binding form is evaluated, the *else* part invokes an error environment. This *else* part should notify the user, telling him which constraint is violated. When the control flow of the program enters the error environment, the programmers can check the values of variables visible to the current lambda expression.

For example, the definition

```
( define next-even (c-lambda (e) (even? e) (+ e 2)))
```

can be transformed into

```
( define next-even
    (lambda (e)
        ( if (even? e)
            ( begin (+ e 2))
            ( break-point "Constraint Violation!"
            '(even? e)))))
```

If the application (next-even 5) is evaluated, the system will trap the error at run-time.

This interpretation does not check if the data constraint expression is functional. Fortunately, interpretations of Scheme+ that check this condition are interesting only if one plans to write Scheme programs that use many assignment based operations.

Another Scheme+ interpretation converts the body of each binding form into simply <expression>+. This reduction is part of a binding form that is at least as defined as the minimal model. This interpretation replaces the condition of the minimal model with the value true and folds the true constant. Since it doesn't check the condition; this model has minimal fault tolerance support. However, it is the most efficient model. One can see that not every interpretation of Scheme must reduce each constrained binding form into an unconstrained binding form with an if expression body.

How can one implement Scheme+ in Scheme? Each constrained binding form can be declared as a Scheme macro. Since most Scheme dialects have a facility that supports user-defined macros, it is not necessary to use a parser generator to write a Scheme+ parser. This theory defines the syntax in a way that expects this strategy. This macro declaration should reduce the constrained binding construct into a special form of unconstrained binding construct which is valid for the Scheme theory.

I built a Scheme+ interpretation for PC-Scheme. I used this system in the case study in the following section. In this interpretation, there is a global variable called debug-mode. This variable determines how the constrained binding form macros will expand. If debug-mode is true, any occurrence of the constrained binding form is expanded to include its data constraint expression as a run-time check. This is like the first Scheme+ interpretation described earlier. If debug-mode is false, the run-time data constraint is omitted.

This is like the second Scheme+ interpretation described.

If one loads a reliable Scheme package, one can set debug-mode to false and produce efficient code. If one is developing or changing a Scheme package, one can set debug-mode to true before loading the package. Then the system will check the constraints.

Ideally a good Scheme interpretation should include the run-time data constraints only when necessary. Too many run-time constraints produce large performance degradation. Too few run-time constraints reduces fault tolerance. When using an implementation that supports run-time constraint checking, the Scheme+ should allow programmers to put checks in some places and remove them from others. Then, he or she can provide the balance of support they need.

One can also build tools to check Scheme+ statically (at compile-time). Such static checking tools must determine if constraints are "purely functional" and "always terminates". Unfortunately these conditions are generally undecideable. It is my hope that an analysis tool can prove most useful constraints have these properties.

One can see that the Scheme+ theory is defined in terms of Scheme. Also, developers can define the constructs of a Scheme+ interpretations as changeable parts. Since there is no need to extend Scheme's framework with foreign constructs, this language extension should be easy for Scheme programmers to understand. Furthermore, the abstract nature of the Scheme+ theory will allow developers to define many interesting Scheme+ interpretations.

9. A Case Study of Scheme Data Constraints

This section illustrates the use of this constraint checking method to build a large, natural-language processing system. This section describes a simplified version of a system written in a Scheme+. The Scheme+ implementation used allows one to use run-time constraint checking facilities during development and turn them off in the finished product.

Each module takes as input a grammar for a subset of English, a grammar component (something that occurs on the right-hand side of a production from the given grammar), and a suffix of the original input sentence.

Each module returns an abstract variant record called conditional result. The tag of the conditional result is a Boolean. It indicates whether the parser recognizes some prefix of the sentence parameter (with respect to the grammar and the grammar component). If the tag is true, the conditional result contains successful parse data. This consists of a syntax tree with semantic information and the suffix of the given string which hasn't been processed. If the tag is false, the conditional result contains some value to relay this fact.

The designer uses data constraint annotations to describe the domain and the range of these functional modules. The description of the domain and range has the usual abstract type (context-free grammars, grammar components, strings, and syntax tree with semantic information). There are familiar records with known structure (conditional results and successful parse data).

The definition of a grammar component is broad. To build a natural language processor, one should refine the grammar component into partitions. Then, one can build a handler for each specific form of grammar component. Scheme partitions its domain of expressions into atoms and lists of expressions. Therefore, it is reasonable for a natural language processor written in Scheme to partition grammar components into symbols and sequence of symbols. Furthermore, symbols can come in two varieties: terminal and non-terminal. Non-terminal symbols should have rules that will provide alternate ways to expand them. Therefore, the modules for a natural language processor could be

    1a. parse terminal symbol
    1b. parse non-terminal symbol
    2. parse symbol sequence

In additions to types, the description of the domain and range of the program defines relationships. The input of a module can't be any grammar component; it must be one that occurs on the right-hand side of a production from the given grammar. The input can't be any string; it must be a suffix of the original string passed to the natural language processor. So given data constraints, one can describe these relationships.

As developers create Scheme data constraints to document relations, they get a better understanding of the module interface. For example, when the grammar component passed to a parse symbol sequence module is an empty sequence, the suffix sentence of any successful parse data passed back is also empty. Therefore, it is incorrect to say that the suffix sentence must be a *proper* suffix of the input sentence.

During program testing, the debug-mode flag is set before the program loads into the interpreter. So, the data constraints help one to localize trivial errors, like putting actual parameters in the wrong order. For example, given a procedure with the form

```
( lambda ( a b )
        ( and ( integer? a ) ( integer? b ) (<? a b )) ...
)
```

this interpretation of Scheme+ will signal a run-time error if a program applies the actual integer parameters to this procedure in the wrong order.

The development version of the program is slow and space inefficient. When one is confident that one has eliminated the trivial errors, using program verification and program testing, one turns debug-mode off and compiles the modules.

As the system evolved, the constraints on particular aspects of the system changed. For example, one might change the domain of a procedure from proper lists to non-empty proper lists. If a program still passes an empty list to the procedure, after one has made the proposed change to the program, the system will catch the error. User defined constraints helps one to find out how changes violated old constraints.

One can make the following observations.

1. If a data constraint cannot be true, then one would want it to be false (it is possible for a constraint to raise an error or loop forever). This is the case because some Scheme+ interpretations handle false constraints elegantly.

2. Data constraints have the potential to be verbose. At the extreme, documenting every possible data constraint makes the program look busy and detailed. If the system checks all constraints dynamically, the constraint will cause the program to run unnecessarily slow. A developer must use his intuition to determine when a constraint is necessary. For example, the expression (c-expr r (zero? r) 0) has a constraint that is unnecessary. It is always satisfied and it doesn't give the reader any more information than the object

expression 0. For simple examples of polymorphic functions, a user or a machine can look at the procedure body and induce a constraint for the domain and range. For example, given the expression

(lambda (x) (+ x 7)) ,

one can see that x must be some number. If a tool could *induce* data constraints from Scheme programs, the programmer could put in less verbose constraints. The programmer would put in the interesting constraints that a reader or a machine could not induce on its own.

For example, consider the expression (<? a b) . This expression would not be a good data constraint because if a or b is not a number, then it will raise an error. But since the variables must be numbers to make the constraint true, a tool may be able to induce the improved data constraint (and (number? a) (number? b) (<? a b)) from the original.

But, developers should put in data constraints when they feel confirmation of the type inference will help.

It should be noted that strongly typed languages are also verbose, but they provide fewer opportunities to avoid verbosity. In the following declaration, the required type mark STRING does not provide any more information.

A : **constant** STRING := "Hello World";

3. Data constraints help one to understand one's module interfaces. This method makes orthogonal languages more feasible for general-purpose programming. Since orthogonal languages allow programmers great freedom, it is difficult to detect errors in an orthogonal language program at compile time. This weakness is attributed to their lack of language-defined restric-

tions, which form the basis for static checking. But user-defined restrictions can also form the basis for static checking.

Data constraints allow the user to put back the structure they want and need.

4. One can observe that it is impossible to write general-purpose decision procedures to determine whether a non-trivial property holds for arbitrary procedure objects. For example, if one applies an arbitrary procedure to another procedure, one might expect the procedure parameter to halt normally if given any type of input. But since this constraint is a statement of the unsolvable halting problem, a programmer cannot document this property with a decision procedure.

This is a problem, because the programmer is trying to use the method in a way that approaches the theoretical limits of computability. Therefore, this is a language-independent problem. This is a problem in any system that supports general data constraints.

In this method, data constraints are decision procedures (they always halt). Therefore, data constraints in this method cannot describe non-recursive sets. (Note: non-recursive sets do not have a decision procedure. It might be possible to allow accepting procedures (that halt only if true) to be data constraints. Then, data constraints can also describe recursively enumerable sets.).

5. Besides this language-independent problem, standard Scheme has one language-dependent problem. Standard Scheme does not provide an explicit way to check if a program applies the correct number of arguments to a procedure object. There is no general way to guard against entering an error environment because a program applies the wrong number of arguments to a procedure. To allow explicit checking for this condition, the Scheme standard should define a new essential procedure that takes a procedure as input. This

procedure would determine the bounds on the number of actual parameters for the input procedure.

Without such a facility, a system can only test this condition implicitly.

From this case study, we can conclude:

1. Data constraints are more flexible than a typing system.
2. Data constraints are more powerful than a typing system in its ability to catch errors at run-time.
3. Describing conditions in terms in "total" decisions procedures are limiting; undecidable properties cannot be documented.
4. Run-time checking can be costly.
5. Constraints can sometimes be verbose and redundant (like a typed language).

10. Performing Static Semantic Analysis on Scheme Data Constraints

This section discusses static semantic analysis of Scheme's data constraints. Motivation for static semantic support is first discussed. Then, this section discusses the tool.

10a. Motivation

Strongly typed languages perform extensive static type checking analysis. But static semantic analysis is not a strong point of Scheme. As an orthogonal language, its lack of rigid structure makes it difficult for a translator to find errors at compile time. This is why Scheme does most error checking at run-time.

Since run-time constraint checking can become costly, deferring all error checking to dynamic semantic analysis is undesirable. For example, a programmer must constrain the domain of the binary search to sorted lists. But, can a system justify using a $log(N)$ procedure just to check the integrity of a binary search parameter? Avoiding the cost of run-time checks was a stimuli for the development of strongly typed systems. These systems eliminate all type errors at compile-time.[5] Though a system can justify some dynamic semantic analysis, it should minimized such cost. Surely, most of the execution time should be spent doing work for the problem itself. Furthermore, run-time constraints can impose a large storage penalty; when a program includes run-time checks, it becomes bigger.

---

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading Mass., page 348.

There is a way to circumvent this problem in a run-time constraint checking method. During development and debugging, developers still impose run-time checks. But before shipping the final product, the developers should convince themselves (through verification and testing) that the constraint checks are unnecessary (always side-effect free and true). Then, they turns off all constraint checking in the product given to the customer. If the constraints are unnecessary, then this will not change the meaning of the program. But because of this decision, a data constraint method cannot help make the finished software product fault tolerant.

Can this design method make Scheme programs easier to analyze *statically*? This is desirable; then Scheme programmers can rely less on testing to debug programs. As explained earlier, static semantic analysis is not feasible in a plain orthogonal language program. But data constraints may put enough structure back into the program to make some static semantic analysis possible. Also, if there are a few simple constraints that an analysis tool cannot remove at compile time, it may be possible to leave the remaining constraint in the program. If an optimizer removes most of the data constraints at compile time, then a program may achieve fault tolerance at a reasonable cost.

Here are simple examples of Scheme constructs that a tool could check at compile time. When a program applies 0 to next-even, one can see (without execution) that the constraint is satisfied. But, when a program applies 1 to next-even, a person can see that the constraint is violated.

10b. The Static Semantic Analysis Tool

To determine if data constraints make Scheme programs easier to analyze statically, I wrote a tool. This tool is meant to perform more static semantic analysis than the Scheme translator. It is much like the C lint facility.

The following are highlights of the lint facility's features.

Since a Scheme lint facility is much like the front end of a compiler, it

has a symbol table. Unlike the symbol table for the Scheme system, the lint facility symbol table stores information about constraints imposed on lambda expressions.

The symbol table stores information about all global variables seen, as well as the variables in local environments. The lint facility adds information about a variable when it processes a local environment. The lint facility removes that information from the symbol table, when it finishes processing a local environment.

This happens because these static scope variables are no longer visible to succeeding environments.

The lint facility was implemented with two symbol tables. One symbol table is for objects and the other is for macros. I made this separation because Scheme does not treat macros as first-class objects. For example, one can't pass macro to procedures or assign macros to different variables. (In the future, it may be interesting to see if a Scheme dialect should treat macros as first-class objects.)

To get information into the lint facility's symbol tables, I built a new loader for my Scheme interpreter. This loader behaves like the system loader, except it puts information into the new symbol tables. For top-level, user-defined procedures, it is enough to store the expected number of parameters, the precondition, and post-condition. By storing the number of parameters, a system can check this condition implicitly.

To analyze Scheme expressions, this lint facility unfolds all macros (both user and language defined) and processes the core language. I made this decision because the lint facility was meant to be a stand-alone tool (as opposed to being integrated into the translator). As a stand-alone tool, the lint facility and the Scheme translator perform redundant analysis. But by unfolding macros, the lint facility's can avoid some redundant processing. Also, it

seemed possible to decompose an expression, analyze it, and restore the high-level form elegantly for diagnostics. And if this is not possible, it is still possible for a human to find errors in a code segment, when given the equivalent, decomposed, core language code segment.

However, as a result of processing the core language, the analysis of programs with call/cc (a mechanism used to manipulate continuations explicitly) is difficult. At this level, the call/cc in Scheme is like a goto statement in an imperative language; it is difficult to specify formally. The call/cc mechanism is easier to specify at a lower level of abstraction.

The lint facility expects all external definitions (global variables) to be explicitly declared. Since the Scheme's formal semantic description does not explain external definitions, translators treated them differently from internal definitions (those in a local environment). Furthermore, different Scheme implementations treat external definitions inconsistently. Some versions of Scheme allow set! to declare top-level variables implicitly. These versions treat define as another name for set!. Other versions enforce the property that a program should explicitly declared variables with define, before that program can assign values to its variables. In deference to good software engineering principles, the lint facility expects explicit declarations. A lint facility that allows set! should warn people when their program has not defined the assignee.

The Scheme lint facility was implemented in MacScheme. Though the case study was developed using PC Scheme, I thought it would be interesting to show that the data constraint method was not implementation specific.

The lint facility has one advanced feature; it doesn't stop on the first error. When an error is detected, the tool records the error, invokes error recovery, and looks for more errors.

While building the lint facility, many problems were encountered.

Looking beyond the error recovery, the lint tool created is primitive. It can tell the programmer when he is using the wrong format for a basic form. For example, the lint tool records an error if a basic form has the wrong number of parameters. The lint facility can tell the programmer if a symbol has not been bound. It can update the symbol table when it sees external definitions and macros.

Unfortunately, it is not sophisticated enough to perform sophisticated program verification. Also, this version of Scheme lint facility does not build libraries.

Furthermore, during the development phase, memory problems were encountered; It is difficult to load the lint tool and use it on a Macintosh SE system with 1 megabyte of RAM.

Nevertheless, I am confident that a designer could take this primitive tool and produce a program that can perform extensive static semantic analysis on Scheme programs.

Furthermore, I believe that an orthogonal language like Scheme supports the data constraint method better than a strongly typed procedural language. Since the data constraint method makes it possible to put the structure back into an orthogonal program, it is possible to track errors in an orthogonal language, without sacrificing flexibility. This makes orthogonal languages feasible for general-purpose programming.

11. Plans for the Future

Eventually, I would like to use a theorem prover, which uses unification, to verify constraints at compile time. Unification has been successful in determining the type inferences of polymorphic functions in the ML program language.[6] It is my hope that a developer can build a type inference system for lambda expressions and extend it so it can also confirm user-defined data constraints.

I would also like to integrate this improved facility into the phases of some Scheme language translator. This facility will need privileged read/write access to the system symbol tables. This will reduce the redundant work that a stand-alone tool must perform (like processing macros and maintaining a separate symbol table). This makes it possible to provide the extensive static semantic support with less space penalties. Though the tool requires privileged access to the symbol tables of a Scheme translator, this will not change the language to the end user. Also, the new facility will need a library, so it can remember information about previously processed external definitions. Furthermore, the decisions on the status of assignment statements and external definitions will still apply.

---

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools,* Addison-Wesley Publishing Company, Reading Mass., page 364-70.

regular Scheme
Scheme without user macros
core-language Scheme
Scheme in continuation passing style
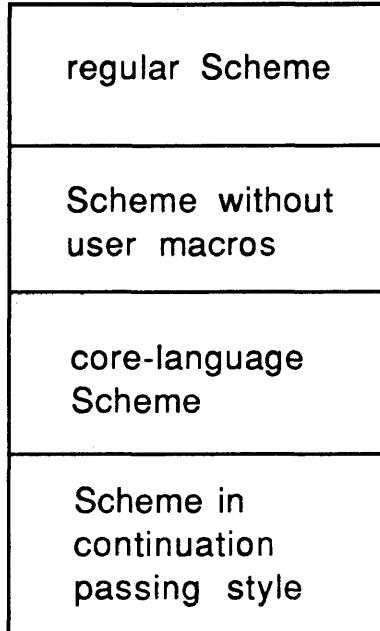
Originally, I decided to analyze data constraints at the core-language level. Now, I would explore analyzing data constraints after the translator converts the program into continuation passing style. At this point, denotational semantics (not Scheme) describes the meaning of the program.

Beside creating an improved static semantic analysis tool, computer scientists should determine the feasibility of functional and function-oriented languages for general-purpose, production quality programming. What are the consequences of eliminating the destructive assignment statement? Are problems harder to specify without assignment statements? Are functional programs innately less efficient? How much harder is it to process function-oriented languages versus functional languages? What is the best medium for the end users? I would hope that Scheme can provide more features that will reduce the need to use assignment statements further, without making the language rigid or more complicated.

To simplify the user's view of the Scheme language further, we need more research in exploiting unbounded integers and rational numbers (with correct precision). Unlike fixed point and floating point numbers, these types

should behave more like mathematical numbers and have simple axioms. The revised report on Scheme encourages the support of such types; "What the user should see is the usual operations on numbers produce the mathematically expected result, within the limits of the implementation."[7] Unfortunately, the Scheme dialects I have tested appears to use the conventional fixed and floating points. The unbound integers and rational numbers should allow more flexibility for a translator to perform extensive algorithm improvement.

It was suggested earlier to consider the worth of using data constraints for non-recursive sets (may not halt on bad data). If such a constraint is violated, run-time error checks may not halt. However, these types of data constraints, though highly fault intolerant, might still be useful for increasing one's ability to document problems and localize errors during debugging.

In the future, the Scheme language definition should introduce a construct called `define-constant`. One would use this to declare read-only, external definitions.

Lastly, one lesser question that needs to be answered is "should Scheme treat macros as first-class objects?"

Obviously the data constraint method is well-suited for function-oriented languages. Function-oriented languages can support the method better than procedural languages. Introducing data constraints into languages that embrace the von Neumann state transition model makes these languages too complicated. Function-oriented languages are generally more feasible to use than functional languages. Though it is harder to detect errors in a plain orthogonal language, data constraints can make function-oriented languages easier to analyze than a strongly typed language. Though dynamic semantic analysis is costly, there is promise that software tools can check data con-

---

[7] Rees, Jonathan and William Clinger, ed. *Revised³ Report on the Algorithmic Language Scheme*, MIT Artificial Intelligence laboratory, September 1986, page 18.

straints   statically.

Appendix: An Introduction to Scheme

This section is a gentle introduction to the features of Scheme relevant to this thesis. First, this section will highlight some features of Scheme. Then, this section discuss what one can do in Scheme. Lastly, the formal semantic definition of Scheme will be discussed.

The programming language Scheme is derived from Lisp. Because it is easily extended and modified, Lisp has served as the basis of many attempts to create a better language.

> Only the compiler writer can change Pascal, but any Lisp programmer can alter Lisp, or adapt it to some specific purpose, or even build an entirely new language on top of it.[8]

Superficially, Scheme has the uniform, simple syntax of Lisp. A symbolic expression is either an atom or a list of expressions delimited by parenthesis. All function calls have prefix operator notation.

Scheme has both its programs and its data using the same written format. This simplifies treating programs as data. Like other forms of data, programs can be read, written, or built by other programs.

Scheme has automatic heap management and encourages recursive thinking.

---

[8] PC Scheme Tutorial, Texas Instruments, Austin, Texas, page ix.

Scheme maintains Lisp's tradition of simplicity. Scheme has a formal semantic definition whose framework has a core language that includes constants, variables, procedure applications, lambda expressions, "if" conditional expressions, and "set!" assignment expressions. The Scheme definition uses this lean core language to describe other concepts in the language.

Scheme variables have static scope. The following code illustrates this fact:

```
==> ( define a 5 )
a
==> ( define ( add5 x ) ( + a x ))
add5
==> (define ( f y ) ( let (( a 3 )) ( add5 y )))
f
==> (f 6)
??????
```

Intuitively, programmers familiar with block structured languages would expect (f 6) to return 11. The preceding segment does just that, because these variables have static scope. The variable a in the function add5 refers to the global variable, because add5 was *declared* in the scope of the global environment. But in a language where variables have dynamic scope, (f 6) will return 9. Assuming dynamic scope, add5 would *evaluate* variable a in the scope of a local environment that hides the global a with a local a. A static scope variable is context-free; an identifier within a subprogram will refer to the same object, irrespective of where that subprogram is called. This is not the case with a dynamic scope variable. This is probably why static scope is more popular and easier for programmers to comprehend.

Almost every Scheme entity is a first-class object. This makes Scheme

orthogonal; one can manipulate procedures in the same way as integers and strings.

In Scheme, only one value is associated with an identifier. Alternately, some lisp dialects allow a variable within the scope of some program to be a function in one place and an object in another place. These dialects do not treat procedures as first-class objects. The Scheme alternative is simpler; the meaning of a variable is context free. The latter alternative adds needless complications for the user and for tool developers.

All Scheme objects have dynamic extent; this means that Scheme objects theoretically exist forever. In actuality, the Scheme system can collect an object as garbage only when all references to that object are lost. The following Scheme procedure exploits dynamic extent.

```
( define count
    ( let (( next 0 ))
        ( lambda ()
            ( let (( v next ))
                ( set! next ( + next 1 ))
                v )))))
```

When called in a system without dynamic scope, the procedure count will enter an error environment and complain that it could not find the variable next. In Scheme, this procedure will "hold onto" the variable next from the enclosing local environment.

In a system with objects that have dynamic extent, the stack model of allocating local environments is abandoned for the heap model. In the previous example, the expression associated with the environment containing that local variable next has executed. In a stack-based system, the lifetime of that variable is completed and its object is collected. In a heap-based system, the object of that variable is not collected, because there is an existing

reference to that variable.

This example shows how dynamic extent eliminates the possibility of a form of dangling reference.

And, the feature allows the preceding procedure to have state data that is not in the global environment. The following example shows how the state based procedure count works.

```
==> (count)
0
==> (count)
1
==> (count)
2
```

There is no chance that another procedure can overwrite this state data accidentally. The visibility of next is restricted to procedure count.

This feature of Scheme makes the language capable of supporting packaging mechanisms and message passing.

Scheme has efficient recursion (no needless storage penalty for using tail recursion). By abandoning the stack model of allocating local environment, Scheme presents a model of computation that may seem less efficient to the user. But Sussman and Steele, the creators of Scheme, found that it is reasonable to optimize Scheme programs so some programs allocate local variables on a stack or in registers (this will be explained in more detail shortly). So though the user can think that Scheme allocates all objects from the heap; the translator may also store objects in a stack or in registers. The user doesn't have to worry about these details; he or she can concentrate on writing a program in this simple, high-level, one-layer environment.

Scheme has powerful first-class objects called continuations. They represent the future of a computation. They are like the address in a program

counter, except continuations contain *all* the information needed to continue a particular computation.

Continuations give a name to a concept that already exists in programming languages. Computer scientist are aware that program statements specify a particular flow of control. After the condition of an if statement is evaluated, control flow resumes at either the *then part* or the *else part*. Scheme gives advanced programmers a hook so they can specify control flow explicitly.

It is noteworthy that continuations are a form of an unstructured branch, yet continuations do not have some of the problems associated with goto statements. Advocates of structured programming know that goto statements do not work well within the framework of a stack-based environment. But the continuation (an unstructured construct) works well in a heap-based environment (with dynamic extent). Furthermore, programming languages with continuations can provide powerful extensions, such as coroutines and non-blind backtracking, within the language framework. Since programming languages have an influence on a programmer's problem solving approach, continuations will allow programmers to be more creative.

How is Scheme used? Scheme is flexible enough to support many program paradigms. Scheme can support manifest types (strong typing achieved by using tags), delayed evaluation, normal-order evaluation, packaging, generic operators, logic programming, deductive information retrieval, and message passing. Likewise, Scheme is suitable for supporting new paradigms.

Scheme is so simple, it is a good candidate for formal semantic definitions. With Scheme, one can show how to define program semantics using text substitution, the conventional environmental model, a Scheme

interpreter written in Scheme, translation to primitive register machines, and denotational semantics.

Scheme has a standard language definition called the <u>Revised Report of the Algorithmic Language Scheme</u>.

The standard includes a language framework with a denotational semantic definition that uses the Scott-Strachey approach to programming language theory. This is beneficial, because denotation semantic definitions help language experts understand the fine details of a language feature. Designers define languages like Ada using English prose. As a consequence, the fine details of some language features are unclear, because its English prose description is ambiguous.

The standard also includes a section that describes compound language features as macros. These macros reduce these features into core language code segments.

The denotational semantics of Scheme embraces the continuation passing style, where control flow is expressed explicitly using continuations. As a benefit, the formal semantic definition of Scheme is small, simple, and flexible, when compared to the denotational semantics of other languages.

As a benefit, this style makes the language definition easier to optimize. This is why Scheme translators can support efficient recursion. It is also why the developers of ORBIT, an optimizing compiler for Scheme, have produced object code that is competitive with object code produced by compilers for conventional Algol-like languages. Even without performing conventional optimizations, like common sub-expression elimination and code motion (moving redundant code outside loops), the Orbit compiler produced code that is comparable to the best compilers for Lisp and non-Lisp languages.[9]

Though all Scheme dialects must adhere to the standard, some

---

[9] David Kranz, "ORBIT: An Optimizing Compiler for Scheme," SIGPLAN, page 219-22.

procedures in the standard are non-essential. The standard only encourages developers to build a macro facility; it does not call for every Scheme dialects to have one. Also, the standard does not impose a uniform macro declaration and macro application method for every Scheme dialects. The standard is flexible on this issue, because it feels there is no universally superior macro facility model.[10] Furthermore, since the definition vaguely describes some language features, different dialects came up with different interpretations.

As it stands, Scheme is a language that is simple, yet flexible enough to support many paradigms and suitable for large scale production quality programming.

---

[10] Rees, Jonathan and William Clinger, ed. *Revised³ Report on the Algorithmic Language Scheme*, MIT Artificial Intelligence laboratory, September 1986, page 37.

BIBLIOGRAPHY

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading Mass.

Allen, John R., "The Death of CREATIVITY: Is Common Lisp a Lisp-like Language?",*A I Expert*, February 1987, page 48-61.

Abelson, Harold and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, McGraw-Hill Book Company, New York.

Barstow, David R., ed., *Interactive Program Environments*, McGraw-Hill Book Company, New York, 1984.

Bauer, F.L., *The Munich Project CIP Volume 1: The Wide Spectrum Language CIP-L*, Springer Verlag, 1985.

Comprehensive Examination Question List, The College of William & Mary, 1989.

Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, second edition, Springer-Verlag, Berlin, Germany, 1984.

Davis, Martin D. and Elaine J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, Inc., Orlando Florida, 1983.

Cheatham, Thomas E. Jr., *Program Refinement by Transformation*, Center for Research in Computing Technology, Harvard University, Cambridge, MA.

Darlington, John, Program Transformation, *Byte Magazine*, August 1985.

Dybvig, R. Kent, *The Scheme Programming Language*, Prentice-Hall, Inc. Eglewood Cliffs, New Jersey, 1987.

Gries, David, *The Science of Programming, Springer-Verlag*, New York, 1981.

Hofstadter, Douglas, *Goedel, Escher, Bach: An Eternal Golden Braid*, Vintage Books, New York.

Horowitz, Ellis, *Third Edition Programming Languages: A Grand Tour*, Computer Science Press, USA, 1987.

Kranz, David, "ORBIT: An Optimizing Compiler for Scheme," SIGPLAN.

Luckham, David C., *ANNA : A Language for Annotating Ada Programs Reference Manual*, Springer Verlag, Berlin, Germany, 1987, page 1.

MacScheme + Toolsmith^{TM} a Lisp for the future, Semantic Microsystems, Beaverton, Oregon, February 1988.

McCarthy, J., A Basis for a mathematical theory of computation., *Proc. Western Joint Computer. Conference.,*Los Angeles, May 1961, page 225-238, and *Proc. IFIP Congress 1962*, North Holland Publ. Co., Amsterdam, 1963.

Mills, Harlan, The New Math of Computer Programming, *Communications of the ACM*, volume. 13, number 1, January 1975, page 43-48.

PC Scheme A Simple, Modern Lisp User's Guide, Texas Instruments, Austin, Texas.

PC Scheme Tutorial, Texas Instruments, Austin, Texas.

Rees, Jonathan and William Clinger, ed. *Revised^3 Report on the Algorithmic Language Scheme*, MIT Artificial Intelligence laboratory, September 1986.

Stoy, Joseph E., Denotational Semantics: *The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Massachusetts, 1987.

TI Scheme Language Reference Manual, Texas Instruments, Austin, Texas.

VITA

Earl Stavis Harris Jr.

Born in New Rochelle, New York on June 3, 1962. Graduated from Mount Vernon High School in that city, June 1980. Received Bachelors degree from Harvard University, June 1984.

In September 1987, the author entered the College of William and Mary as a graduate assistant in the Department of Computer Science. The following year, he was awarded the Graduate Dean's fellowship. The course requirement for this degree have been completed, but not the thesis: Data Constraints in Function-Oriented Languages.