Djang et al. [1998] introduced a new model of types for declarative visual programming languages (VPLs). Implicit static typing is used in their type model, in order to eliminate the programming mechanisms associated with type declarations, provide immediate visual feedback with respect to type errors and guarantee type safe programs. Their type model also evaluates types on a level of granularity that is finer than in previous approaches to types. Instead of evaluating types on the basis of abstract names, their model determines a set of operations that an object guarantees and compares this set to the set of operations this object is required to support. If the set of required operations is a subset of the set of guaranteed operations, then the object is considered type safe. This granularity provides their model with the ability to support inheritance without introducing explicit type declarations and to communicate type errors to users without requiring the user to understand a large set of terminology. These features of their model attempt to provide VPL users with more powerful programming capabilities without the introduction of a high learning curve.

In this thesis, an implementation of the Djang et al.’s model of types is presented. Data structures and algorithms are developed that conform to the axioms prescribed by Djang et al. The space and time complexity analyses for our data structures and algorithms are examined. Our implementation provides new insights into the cost and performance of the Djang et al.’s type model.
©Copyright by Roger Ding-Fu Chen
September 9, 1999
All Rights Reserved
Guarantees and Requirements:
Implementation and Complexity Analysis of a New Model of Types
for Declarative Visual Programming Languages

by

Roger Ding-Fu Chen

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed September 9, 1999
Commencement June 2000
I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Roger Ding-Fu Chen, Author
Acknowledgement

Praise be to God and Christ for guiding me in my growth and maturation this past year and throughout my life! Appreciation and praise are abound for my mother and father for all their endless love, support and patience. My love also goes out to my brother who has stood strong by my side through these years.

At Oregon State University, this thesis and my academic growth could not have been possible without Professor Margaret Burnett, whom I am especially grateful to have as an advisor. I also thank the past and current members of the Forms/3 research group for providing me with valuable assistance.
Table of Contents

1. Introduction ........................................................................ 1
   1.1 Types................................................................. 2
      1.1.1 A Definition of Types ................................. 2
      1.1.2 The Importance of Types in a Programming Language 3
   1.2 Models of Types.................................................... 3
   1.3 Introduction to Forms/3 ........................................... 5
   1.4 Organization ....................................................... 7

2. Background and Related Works ........................................ 8
   2.1 The Role of Types ................................................ 8
      2.1.1 Type Binding................................................. 9
      2.1.2 Type Checking .............................................. 9
      2.1.3 Type Compatibility ..................................... 10
      2.1.4 Polymorphic Types .................................... 10
      2.1.5 Inheritance ................................................. 12
   2.2 Models of Types .................................................. 13
      2.2.1 Types in Textual Programming Languages .... 15
         2.2.1.1 Static types in the presence of inheritance . 15
         2.2.1.2 Understandability of type inference results . 17
      2.2.2 Types in VPLs .............................................. 19
         2.2.2.1 Static types in VPLs ............................... 20
         2.2.2.2 Understandability of type inference results in VPLs . 21
   2.3 Forms/3 .................................................................. 23
      2.3.1 A Forms/3 Example ..................................... 23
      2.3.2 Core Forms/3: The Subset for Formal Reasoning . 28
         2.3.2.1 Programming objects and notational conventions . 28
         2.3.2.2 Formula syntax and semantics ................. 31
Table of Contents (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2.3 Forms</td>
<td>33</td>
</tr>
<tr>
<td>2.3.3 Translating Between Forms/3 and Core Forms/3</td>
<td>35</td>
</tr>
<tr>
<td>Chapter 3: Djang et al.'s Model of Types—Related Work</td>
<td>38</td>
</tr>
<tr>
<td>3.1 Model of Types: Fine-grained Reasoning in Terms of Guarantees versus Requirements</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Guarantee Sets</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Requirement Sets</td>
<td>42</td>
</tr>
<tr>
<td>3.4 Recursion</td>
<td>46</td>
</tr>
<tr>
<td>3.5 Example: Type Inference Without Inheritance</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Type Inference With Similarity Inheritance</td>
<td>50</td>
</tr>
<tr>
<td>3.7 Example: Type Inference in the Presence of Single Inheritance</td>
<td>53</td>
</tr>
<tr>
<td>3.8 Example: Type Inference in the Presence of Multiple Inheritance</td>
<td>54</td>
</tr>
<tr>
<td>Chapter 4: Design, Algorithms and Complexity</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Data Structures</td>
<td>56</td>
</tr>
<tr>
<td>4.1.1 Representation of Operations</td>
<td>57</td>
</tr>
<tr>
<td>4.1.2 Representation of Sets of Operations</td>
<td>58</td>
</tr>
<tr>
<td>4.1.3 Representation of Primitive Type Operations</td>
<td>60</td>
</tr>
<tr>
<td>4.1.4 Representation of Primitive Form Operations</td>
<td>60</td>
</tr>
<tr>
<td>4.2 Algorithms</td>
<td>61</td>
</tr>
<tr>
<td>4.2.1 Type Inference</td>
<td>61</td>
</tr>
<tr>
<td>4.2.1.1 Type inference for the edited RO</td>
<td>62</td>
</tr>
<tr>
<td>4.2.1.2 Type inference for affecting ROs</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1.3 Type inference for affected ROs</td>
<td>70</td>
</tr>
<tr>
<td>4.2.2 Adding/Removing Operations</td>
<td>71</td>
</tr>
<tr>
<td>4.2.3 Renaming Operations</td>
<td>73</td>
</tr>
<tr>
<td>4.2.4 Type Checking</td>
<td>75</td>
</tr>
<tr>
<td>4.3 Examples of Type Inference and Type Checking</td>
<td>76</td>
</tr>
</tbody>
</table>
Table of Contents (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>Example: Type Inference and Type Checking</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>After an RO is Added to a VADT form</td>
<td></td>
</tr>
<tr>
<td>4.3.2</td>
<td>Example: Type Inference and Type Checking</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>After a Simple Cell Formula Edit</td>
<td></td>
</tr>
<tr>
<td>4.3.3</td>
<td>Example: Type Inference and Type Checking</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>After a VADT Form RO is Renamed</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Space and Time Analyses</td>
<td>79</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Space Analyses</td>
<td>79</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Time Analyses</td>
<td>83</td>
</tr>
<tr>
<td>4.4.2.1</td>
<td>Type inference</td>
<td>83</td>
</tr>
<tr>
<td>4.4.2.2</td>
<td>Adding/removing operations</td>
<td>86</td>
</tr>
<tr>
<td>4.4.2.3</td>
<td>Renaming operations</td>
<td>87</td>
</tr>
<tr>
<td>4.4.2.4</td>
<td>Type checking</td>
<td>88</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Discussion</td>
<td>89</td>
</tr>
<tr>
<td>Chapter 5: Future Research and Conclusion</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Future Research</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Conclusion</td>
<td>92</td>
</tr>
<tr>
<td>Annotated Bibliography</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>Appendices</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 A sample Forms/3 form that contains two cells</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Varieties of polymorphism</td>
<td>12</td>
</tr>
<tr>
<td>2.2 A small program</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Type of function “pair”</td>
<td>18</td>
</tr>
<tr>
<td>2.4 A portion of a Forms/3 form primitive Circle</td>
<td>24</td>
</tr>
<tr>
<td>2.5 A visualization of population data</td>
<td>26</td>
</tr>
<tr>
<td>2.6 VADT form Point</td>
<td>30</td>
</tr>
<tr>
<td>2.7 Translation of the Forms/3 formula</td>
<td>36</td>
</tr>
<tr>
<td>3.1 The Stack form</td>
<td>51</td>
</tr>
<tr>
<td>3.2 The Queue form</td>
<td>52</td>
</tr>
<tr>
<td>3.3 The Remove form</td>
<td>54</td>
</tr>
<tr>
<td>3.4 The Deque form</td>
<td>55</td>
</tr>
<tr>
<td>4.1 Forms/3 RO hierarchy</td>
<td>57</td>
</tr>
<tr>
<td>Table</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>2.1 The grammar for Core Forms/3’s formula language</td>
<td>32</td>
</tr>
<tr>
<td>2.2 Axiomatic semantics for each operator in Core Forms/3</td>
<td>32</td>
</tr>
<tr>
<td>2.3 Semantics of form + (and of forms copied from +)</td>
<td>33</td>
</tr>
<tr>
<td>2.4 Invariant properties of forms</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Every constant value guarantees exactly the operations on its primitive form</td>
<td>40</td>
</tr>
<tr>
<td>3.2 Guarantee sets for some primitive forms</td>
<td>42</td>
</tr>
<tr>
<td>3.3 Requirements sets for some primitive forms</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Type inference algorithms for edited simple cells</td>
<td>63</td>
</tr>
<tr>
<td>4.2 Type inference algorithms for edited matrices</td>
<td>65</td>
</tr>
<tr>
<td>4.3 Type inference algorithms for currently affecting ROs Z</td>
<td>68</td>
</tr>
<tr>
<td>4.4 Type inference algorithms for formerly affecting ROs Y</td>
<td>70</td>
</tr>
<tr>
<td>4.5 Type inference algorithm for affected ROs</td>
<td>71</td>
</tr>
<tr>
<td>4.6 Algorithms for adding and removing guaranteed operations</td>
<td>72</td>
</tr>
<tr>
<td>4.7 Algorithms for adding and removing required operations</td>
<td>73</td>
</tr>
<tr>
<td>4.8 Algorithms for renaming VADT form ROs</td>
<td>75</td>
</tr>
<tr>
<td>4.9 Type check algorithm</td>
<td>76</td>
</tr>
<tr>
<td>4.10 Notational conventions for space and time analyses</td>
<td>80</td>
</tr>
<tr>
<td>4.11 Summary of space analyses</td>
<td>82</td>
</tr>
<tr>
<td>4.12 Worst case time complexities for type inference algorithms</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Tables (Continued)

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Worst case time complexities of algorithms for adding and removing operations</td>
<td>87</td>
</tr>
<tr>
<td>4.14</td>
<td>Worst case time complexities for renaming operation algorithms</td>
<td>88</td>
</tr>
<tr>
<td>4.15</td>
<td>Worst case time complexity for the type checking algorithm</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Appendices

<table>
<thead>
<tr>
<th>Appendix A: Additional Algorithms</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Type Inference</td>
<td>101</td>
</tr>
<tr>
<td>A.2 Adding/Removing Operations</td>
<td>107</td>
</tr>
<tr>
<td>A.3 Renaming Operations</td>
<td>108</td>
</tr>
<tr>
<td>A.4 Time Analyses</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix B: Source Code Examples</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 types3.lisp</td>
<td>112</td>
</tr>
<tr>
<td>B.2 type-check</td>
<td>112</td>
</tr>
<tr>
<td>B.3 type-undo-ref</td>
<td>114</td>
</tr>
<tr>
<td>B.4 type-check-RO</td>
<td>116</td>
</tr>
<tr>
<td>B.5 type-check-RO-deps</td>
<td>118</td>
</tr>
<tr>
<td>B.6 type-check-requiredDeps</td>
<td>119</td>
</tr>
<tr>
<td>B.7 type-check-guaranteedDeps</td>
<td>120</td>
</tr>
<tr>
<td>B.8 axiom-GC</td>
<td>124</td>
</tr>
<tr>
<td>B.9 axiom-GM</td>
<td>125</td>
</tr>
<tr>
<td>B.10 axiom-GAprime-add</td>
<td>126</td>
</tr>
<tr>
<td>B.11 axiom-GAprime-remove</td>
<td>127</td>
</tr>
<tr>
<td>B.12 axiom-R1a</td>
<td>128</td>
</tr>
<tr>
<td>B.13 aximo-R1bc-add</td>
<td>129</td>
</tr>
<tr>
<td>B.14 aximo-R1bc-remove</td>
<td>130</td>
</tr>
<tr>
<td>B.15 axiom-RM</td>
<td>131</td>
</tr>
<tr>
<td>B.16 axiom-RN</td>
<td>132</td>
</tr>
<tr>
<td>B.17 Implementation Details</td>
<td>132</td>
</tr>
</tbody>
</table>
List of Appendix Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Type inference algorithms</td>
<td>101</td>
</tr>
<tr>
<td>A.2 Algorithms for adding and removing operations</td>
<td>107</td>
</tr>
<tr>
<td>A.3 Algorithms for renaming operations</td>
<td>108</td>
</tr>
<tr>
<td>A.4 Summary of worst case time complexities for additional type system algorithms</td>
<td>111</td>
</tr>
</tbody>
</table>
Dedication

Consider it pure joy, my brothers, whenever you face trials of many kinds, because you know that the testing of your faith develops perseverance. Perseverance must finish its work so that you may be mature and complete, not lacking anything.
—James 1:2-4
Two of our objectives in the development of visual programming languages (VPLs) include: the simplification of the programming process and improving program understandability [Burnett 1991]. The concept of types, however, appears to conflict with one of these objectives because types complicate the programming process by adding new programming mechanisms. The difficulty with eliminating types is that programs may be type incorrect, and program end-users may encounter run-time program errors.

Most VPLs address this type issue by eliminating explicit type declarations, thereby concealing the type system from the user. In these VPLs, dynamic typing often is performed. That is, types are determined at run-time. The problems with this approach include: feedback on type errors occurs during program run-time and not program-entry time, some type errors may not be discovered, and program users instead of program developers will be the recipients of the type error messages. In other VPLs, static typing is used, where static typing refers to a system that determines types prior to run-time. The type systems in these VPLs, however, often encounter other problems.

Another issue associated with types is that support for inheritance results either in additional programming tasks or in the support for some but not all forms of inheritance. In most languages that support implicit static typing, the presence of inheritance results in the addition of explicit type declarations. This result once again complicates the programming process. In most statically typed VPLs, inheritance is not supported. In all textual languages and VPLs, similarity inheritance [Djang and Burnett 1998] is not supported because each language’s type
model evaluates types on the granularity of names or structures. This level of granularity is too coarse to support similarity inheritance. Djang et al. [1998] present a theoretical model to solve this problem. Prior to our work, however, no implementation of this model had ever been attempted.

In this thesis, we present an implementation and complexity analyses of Djang et al.’s model of types. The advantages of our system include: programs are guaranteed to be type-safe because all type errors are discovered at program-entry time, users are required to understand only a small vocabulary of types, and users do not encounter the complex type concepts in some current languages such as union types and function types. Inheritance including similarity inheritance is supported without the introduction of explicit type declarations by evaluating types on a different level of granularity. An object is not type checked using abstract type names. Instead, an object is considered type safe if the operations the object is required to support are a subset of the operations the object guarantees it can support. Our implementation was prototyped in the research spreadsheet VPL Forms/3 [Atwood et al. 1996; Burnett and Gottfried 1998].

1.1 Types

1.1.1 A Definition of Types

In typed programming languages, a type is an abstraction, representing a set of values and behaviors defined for a particular programming abstraction. A variable contains an instance of an abstraction and can be described by its type. For example, in some typed languages, if a variable has the type integer, then this variable can retain a value in the range of -32,768 to 32,767 and be used in arithmetic operations such as addition and subtraction. This variable could not be assigned the value “hello”. Types are used to impose a set of constraints on different abstractions, in order to ensure that instances of these abstractions in a running program behave and interact correctly.
1.1.2 The Importance of Types in a Programming Language

Untyped languages also support abstractions that are the equivalent of types. In these languages, however, the mapping between a high-level abstract concept and its actual usage may be more difficult to determine than in typed languages. User understandability is sacrificed in favor of other representations of data abstractions. For example, in λ-calculus, all data structures are represented by functions. The type integer would be represented by a function instead of a collection of values and behaviors.

In untyped languages, however, the concept of types still exists because it is a natural occurrence in an untyped universe. Cardelli and Wegner [1985] write:

As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

The advantage of a typed language over an untyped language is that for end-users types may be a more intuitive concept than λ-calculus or set theory. End-users, however, have no formal training in type theory, and any programming language designed for or used by an end-user audience has to address this issue in its model of types.

1.2 Models of Types

Typed languages differ from one another with respect to several issues. For example, the type of a variable can be determined before the program is run (also known as static typing) or while the program is running (dynamic typing). The type
of a variable also can be explicitly declared by the user (explicit typing), or it can be inferred by the system (implicit typing).

One of the difficulties in introducing types into VPLs is that enforcing a type safe program may introduce new programming mechanisms, or it may require users to understand complex information. For example, some statically typed languages that support inheritance require users to explicitly declare types. Explicit type declaration introduces additional programming mechanisms that otherwise could be eliminated through implicit type declarations. Some languages support complex types such as union types and function types. These complex types introduce new concepts to the user.

In order not to introduce any new concepts or mechanisms to VPL users, most VPLs have been designed with implicit dynamic typing. The result, however, is that programs may be written that are not type safe. These programs may produce run-time type errors which are displayed to the program's end-user.

The problems with these other dynamically typed models lie in the fact that these models use dynamic typing. Dynamic typing allows error-prone programs to be written. These type errors are discovered and displayed for a user while a program is running.

In the work of Djang et al. [1998], a new theoretical model of types was presented that addresses these issues. Static typing is used to detect program-entry type errors. Some programming mechanisms are eliminated by supporting implicit type declarations. In order to support inheritance without the introduction of explicit type declarations, types are not defined by names or structures. Types are evaluated on the basis of the operations they support. That is, a type guarantees that any instance of it can perform any member operation in a set of operations. When a variable is referenced in an operation, it is required to support a certain sets of operations. Type safety is enforced and checked by making sure each variable's set of required operations is a subset of its guaranteed operations.

For example, given the operation "x + 3", the "+" operation requires that the variable x support the set of operations defined for a number type. If \( x = 2 \), x has the
type number, and x is guaranteed to support the operations defined for number types. In this case, the program is type-safe. If x were to equal "hey", then the program would not be type-safe because x would not support number type operations.

1.3 Introduction to Forms/3

In this thesis, Djang et al.'s model of types was prototyped using Forms/3. A brief introduction to Forms/3 is presented in this section.

Forms/3 is a research spreadsheet VPL. A Forms/3 program is a collection of one or more forms, each of which is the equivalent of a spreadsheet. Each form consists of a set of cells which are created by the programmer through direct manipulation (Figure 1.1). Unlike most commercial spreadsheets, cells in Forms/3 are not set in a grid. Each cell then is defined by a formula. The formula can be typed in or graphically entered through a sequence of direct manipulations. The cell formulas on a form make up that form's calculations.

Forms/3 is both declarative and responsive. Forms/3 is declarative because cell formulas are specified through references to other cells. In other words, a Forms/3 program is defined by the data relationships between the program's cells. Forms/3 is responsive because immediate visual feedback occurs: after a cell is given a formula, the cell's value is calculated and displayed immediately.
Figure 1.1. A sample Forms/3 form that contains two cells. cell1's value "12" is displayed in cell1's interior region, whereas its formula "7+5" is displayed on cell1's bottom-right corner. cell2's value is a graphical circle. Its formula is "circle 15" where 15 is the value of the circle's radius. Each cell's formula can be displayed and undisplayed.

In Forms/3, types are defined by a type definition form. A type's range of values and set of behaviors are defined by the cells on the type's definition form. Forms/3 contains a set of built-in types such as number and circle types. Forms/3 also supports user-defined types. A user can define his/her own type by using a type definition form and providing the form with cells and formulas.

Prior to our work, the model of types in Forms/3 included implicit static typing. Implicit static typing guaranteed that programs were free of type errors and that programmers did not have to declare types. Type checking, however, was performed on the granularity of type names. The problem with this approach is that similarity inheritance could not be supported. In order to support similarity inheritance, a type model that evaluates types on a scale other than names and structures was required.
1.4 Organization

In Chapter 2, work related to types, models of types and Forms/3 is reviewed and discussed. The theory on a new model of types detailed by Djang et al. [1998] is summarized in Chapter 3. In Chapter 4, the implementation of Djang et al.'s model of types is presented and analyzed. The future research discussion and the conclusion follow in Chapter 5.
Chapter 2: Background and Related Works

Incorporating a type system into a VPL, at first, appears to conflict with our VPL design goal of simplifying programming. Some VPLs attempt to conceal the type system from the user by using implicit dynamic typing. The problem associated with this approach is that type-incorrect programs can be produced. If a program is type-incorrect, then the program’s users may encounter run-time type errors. Other VPLs have been implemented with implicit static typing. In these models, types are evaluated on the granularity of type names. The result is that not all forms of inheritance are supported.

In this chapter, the concept of types is briefly discussed. Afterwards, several different models of types will be examined. The chapter concludes with a formal definition of a subset of Forms/3, referred to as Core Forms/3. Core Forms/3 is equivalent to Forms/3, except syntactic sugar and other programming conveniences are eliminated.

2.1 The Role of Types in Programming Languages

Several design issues arise when a type system is incorporated into a programming language. Five issues that arise in designing a type system include: type binding, type checking, type compatibility, polymorphic types and inheritance. These issues often are related to how a variable’s type is determined, how an expression is checked for type correctness, and how differing types are related to one another. Each of these issues affects the understandability, usage and capabilities of a language’s type system.
2.1.1 Type Binding

The issue of type binding involves two design decisions: 1) how is the variable’s type determined?, and 2) when is a variable bound to a type? These decisions affect a language’s programming difficulty and understandability.

Explicit type declaration exists when a programmer has to state the type of each variable used in a program. In a language with implicit type declaration, a programmer does not have to state the type of each variable used. Each variable’s type can be inferred. This inference is referred to as type inference. Several different type inference algorithms are used. These algorithms will be reviewed in Section 2.2.

Type binding can occur either statically or dynamically. If type binding is static, a variable is bound to its type before program run-time, and its type cannot change during run-time. If type binding is dynamic, a variable is bound to its type during run-time, or its type can change during run-time. A variable’s type is determined when the variable is assigned a value. That is, when an expression is executed, the left-hand side variable’s type is assigned the type of the value of the right-hand side.

2.1.2 Type Checking

Type checking is performed on the operands of an operator. In order for a program to be type safe, the types of each operator’s operands must be compatible with the operator’s requirements for that operand. For example, given the expression $x + y$, $+$ is the operator, and its two operands are $x$ and $y$. Since $+$ is an arithmetic operation, its two operands are required to be number types. If $x = 5$ and $y = 6$, then the expression is type safe. If $x = 5$ and $y = “hello”$, then the expression contains a type error.
If a type system uses static typing, then type checking can be performed statically. In some situations, static type checking may not be possible. For example, if a language allows a variable to consist of various types (e.g., unions in C), then dynamic type checking must be performed in order to determine the type of a variable at a particular point in time. In a dynamic typing system, however, type checking must be performed dynamically.

2.1.3 Type Compatibility

The issue of type compatibility arises from type checking. When type checking is performed, what constitutes a compatible type? This issue is important because this design decision affects a programmer’s usage of data in operations. Sebesta [1996] writes: “The design of the type compatibility rules of a language is important, because it influences the design of the data types and the operations provided for objects of those types. Perhaps the most important result of two variables being of compatible types is that either one can have its value assigned to the other.”

Types are compared in two different manners. First, types can be equivalent if they have the same name. This compatibility is referred to as name equivalence. The second form of comparison is structure equivalence. Structure equivalence states that two types are equivalent if they share the same structure.

2.1.4 Polymorphic Types

A language that supports polymorphic types allows variables or function parameters to be bound to different types during program execution. Two major forms of polymorphism include ad-hoc polymorphism and universal polymorphism. Ad-hoc polymorphism refers to a function that operates on different types, but for each type, the function may have a different behavior. Ad-hoc polymorphism is
divided into two subcategories of polymorphism: overloading and coercion. Overloading refers to functions that operate on different types, but for each type, its respective function executes code that is different from the other types. A common example of overloading is the + operator. In some languages, the + operator can operate on number types and text types. For number types, the addition operation is performed, but for text types, text concatenation is performed. Coercion affects variables. Coercion occurs when an argument is converted into a type that is expected by its function. An example of coercion is 3 + 4.5. The + operator expects to add two integers or two floating point numbers. In this situation, the type system in most programming languages coerces the integer 3 into a floating point number.

Universal polymorphism is a general term for a set of types that share the same code. Universal polymorphism can be divided further into two categories: parametric or operation polymorphism, and inclusion polymorphism. Operation polymorphism describes a function that executes the same code for different types. Inclusion polymorphism is used to model the subtypes and inheritance schemes commonly found in object-oriented programming [Cardelli and Wegner 1985]. Subtyping is an example of inclusion polymorphism.

Universal polymorphism often is contrasted with ad-hoc polymorphism [Cardelli and Wegner 1985]: “In terms of implementation, a universally polymorphic function will execute the same code for arguments of any admissible type, whereas an ad-hoc polymorphic function may execute different code for each type of argument.” Figure 2.1 presents the different major forms of polymorphism.
2.1.5 Inheritance

In object-oriented programming, a hierarchy of class relationships can be created. A class can be subclassed under another class. This relationship often is referred to as a “is-a” relationship. For example, a dog “is a” mammal. Inheritance is an object-oriented term that refers to a child class’ ability to access data and behavior associated with its parent class. This section provides necessary background in similarity inheritance for showing in ensuing sections how the model of types can be extended to support similarity inheritance, and thereby to support traditional forms of inheritance as well.

Similarity inheritance [Djang and Burnett 1998] is a new, fine-grained model of inheritance intended for responsive, declarative VPLs. From the perspective of type theory, the most important difference from other forms of inheritance is the fact that it allows inheritance of not only entire types but also individual operations. From the perspective of the VPL user, another important difference from traditional forms of inheritance is that similarity inheritance’s basic relation ("like") is about implementation similarities only, instead of traditional inheritance’s “is-a” relation, which is essentially about similarities between different abstractions. For example, with similarity inheritance, it is sensible to say that a Queue is “like” a Stack, in order to share some of the implementation. With traditional inheritance, however,
sharing code in this particular way would be considered to be bad design (since a
Queue is not a Stack in an abstract sense). Similarity inheritance is intended for
users who are not trained in object oriented programming. It supports their reuse of
code by allowing them to customize starting from any existing example, as in
copy/paste, but without the copy/paste disadvantage of losing all the underlying
relationships.

2.2 Models of Types

The goal of our research is to develop a model of types for declarative VPLs that
support inheritance. The previous section, however, has shown that many different
kinds of models can be constructed. Burnett [1991] suggested five goals for a
declarative VPL type system:

1. The type system supports incremental type-checking. This allows
   feedback to be given immediately if the user enters a formula that
causes the program to become type-incorrect.
2. The concreteness of the programming environment and user’s
   conceptual model is preserved. By this we mean that the user does
   not have to think abstractly about types, and that the type system
   should not require new concepts of the user.
3. Any type errors result in immediate and meaningful feedback. A
type error must follow accepted rules for good error reporting, namely
exactly what is wrong, and where it is wrong. In order to do this, the
system has to use the word “type” in its messages, and refer to the
offending type. From this, we conclude that the type system must not
be invisible. If feedback is to be meaningful, it must refer to types,
and the user must understand types.
4. The type system does not impose new rules on the language that
   would otherwise not be needed. For example, it should not require
declarations, or impose language restrictions solely for the purpose of
supporting the type system.

---

1 Sections 2.2 and 2.3 are excerpts from [Djang et al. 1998] and are the work of the
other authors and me.
5. The type system is general enough for general-purpose programming.

For goals 1 and 3, a static type binding and type checking system is more appropriate than a dynamic typing system because the dynamic type system would not detect type errors until program run-time. The remaining goals suggest that implicit type declarations should be used in a type system because explicit type declarations require additional programming mechanisms. With respect to polymorphic types, Forms/3 supports operation polymorphism. Therefore, operation polymorphism should be supported in the new model of types.

In our model of types, we also decided that the model should support inheritance. Currently, no other language supports similarity inheritance. In order to accommodate inheritance and our two VPL design goals (simplifying programming and improving understandability), we defined the following properties for a declarative VPL’s type inference system [Djang et al. 1998]:

Fine-grained inference: Most static type inference systems derive type information at the granularity of entire classes, and this level of granularity prevents these languages from supporting more fine-grained approaches to inheritance.

Understandability: If a type inference system detects a type error, the error should be communicated to the user. The types in existing models have become so complex that they present difficulties even to professional programmers communicating with the type system. This lack of understandability is not acceptable in VPLs aimed at end users.

Power without the addition of explicit declarations: In implicitly typed languages, the introduction of inheritance or prototyping has typically re-introduced explicit declarations of type constraints.

In this section, type systems relevant to our new model of types are surveyed. These systems are evaluated with the previously mentioned three properties: fine-grained inference, understandability and power without the addition of explicit type
declarations. Most of these languages also incorporate static type binding, static type checking and polymorphic types into their type systems.

2.2.1 Types in Textual Programming Languages

Because the goal of our research has been to develop a type system capable of supporting inheritance in an understandable way, the most closely related works on type systems in textual programming languages are works on static typing related to inheritance and understandability.

2.2.1.1 Static types in the presence of inheritance

Two well-known languages representing the class-based approach and the prototype-based approach respectively are Smalltalk [Goldberg and Robson 1983] and Self [Ungar and Smith 1987]. Although these languages were initially dynamically typed, there is research on incorporating static type inference into both. A type inference algorithm for a simplified Smalltalk that includes inheritance, late binding and polymorphic methods was presented in [Palsberg and Schwartzbach 1991]. The algorithm guarantees that all objects understand all messages sent to them. Self is a prototype-based language that includes both dynamic and multiple inheritance. Like the Smalltalk algorithm, the approach for Self in [Agesen et al. 1993] is to derive and solve sets of type constraints. Both of these approaches handle types on a coarse-grained level, namely at the granularity of classes or prototypes.

Imposing a static view of types on a language with inheritance sometimes leads to problematic theoretical issues. These issues arise from the fact that a fundamental difference exists between subtypes and subclasses [Cook et al. 1990; Harris 1991; Liskov and Wing 1994]. Subtypes reflect the property of substitutability; they should be able to replace supertypes without introducing type errors [Sebesta 1996].
This definition of subtypes allows substitutability of subtypes for supertypes but does not allow overriding in order to specialize a subtype. Subclasses, on the hand, do allow overriding because they are simply an implementation convenience for reusing code and do not inherently guarantee anything about substitutability. The difficulty of combining substitutability with overriding in a type system comes from the difficulty of typing methods whose arguments and return type vary from supertype to subtype.

The solution is to separate the notions of subclass and subtype. In separating these two concepts, the problem of covariance versus contravariance becomes clear. **Covariance** typifies the conventional use of inheritance for reuse; method arguments and results in a subclass are allowed to be subtypes of the arguments and results of the class methods. On the other hand, subtyping requires method arguments of a subclass’s methods to be supertypes (or the same types) as the method arguments of the parent class’s method. This is called **contravariance** because the types of a subclass method’s arguments vary in the opposite way from the method results which are still allowed to be subtypes (or the same types) of the class’s method results. Schwartzbach [1997] succinctly captures the problem’s essence as follows: “for programming purposes [in many cases] we would like to use covariant specialization. However, [without re-type-checking a method in each subclass where it is inherited] only contravariant specialization is guaranteed to preserve static type-correctness”.

Schwartzbach summarizes a variety of proposed solutions to this dilemma, some of which include: supporting only covariance despite sacrificing type safety as in Eiffel [Meyer 1992]; incorporating at least some dynamic typing; and type-checking each method again in every subclass in which it is inherited. Since our approach to type inference is fine-grained, our solution to this problem is most similar to this last approach.

The functional language Haskell [Peterson et al. 1997] has both types and type classes, and this combination provides some inheritance-like characteristics at a finer granularity than traditional classes. Type classes are declarations of a type’s
interface and can also include default implementations of interface methods. A type
class can inherit interface specifications and default methods from other type classes.
A type must implement (or use a default implementation, if one exists) every method
in every type class to which it belongs. While most of Haskell’s type system allows
implicit types which are resolved automatically through unification, explicit
declarations are needed of type classes and of user-defined types’ membership in
them.

None of the languages discussed here provide a type system fine-grained enough
to support similarity inheritance. Most of them reason at the granularity of entire
classes or objects. While Haskell reasons at a finer granularity, namely at the
granularity of interfaces (groups of operations), it does so at the added cost of type
class declarations.

2.2.1.2 Understandability of type inference results

Although the theoretical foundation of implicit polymorphism is rooted in
combinatory logic, Milner was the first to apply the theory to programming
languages [Milner 1978]. Milner’s work has been implemented in many textual
languages, especially in functional languages (such as Haskell, ML and Miranda) in
order to preserve type safety in implicitly typed programs. Within this context, there
has been some work in the functional language community related directly to the
understandability of type inference results. For Milner-based type inference systems,
which reason primarily about functions, understandability of the types is a well-
known problem. One reason for this problem is that when higher-order functions are
present, types may grow exponentially with respect to the size of the program, as
demonstrated in Figure 2.2 and Figure 2.3. Even when no higher-order functions are
present in a program and the types are small, the presence of polymorphic type
variables, type constraints and function types—all of which must be understood by
programmers in order to understand why an erroneous first-order program will not
type check—can be barriers to the acceptance of type inference systems.
fun pair x y = fn z => z x y;
let val x1 = fn y => pair y y in
  let val x2 = fn y => x1 (x1 (y)) in
    let val x3 = fn y => x2 (x2 (y)) in
      let val x4 = fn y => x3 (x3 (y)) in
        x4 (fn z => z)
      end
    end
  end
end;

Figure 2.2. A small program [Schwartzbach 1997].

To address this problem, Wand presented an algorithm to isolate and explain type errors [Wand 1986]. In the algorithm, the two types being compared are each represented by a type tree. A type tree is created by expanding a type variable. When a type variable is expanded, the reason for expansion is saved. In this manner, each tree has a collection of reasons for previous type bindings. When a type error occurs, these reasons are reported. Type error explanations, however, may not be
scalable with respect to program size. For large programs, the two type lists may
grow to be very large. Bent and Duggan furthered Wand’s algorithm by using and
modifying the naive graph unification algorithm used in the Glasgow Haskell
compiler and almost all other ML and Haskell compilers [Bent and Duggan 1996].
Their algorithm adds the ability to handle aliased type variables, but it does not
handle Haskell’s type classes.

Jun and Michaelson presented an approach to improve the ease with which type
errors can be recognized, by encoding types with colors [Jun and Michaelson 1999].
This color visualization approach has been implemented in a visual environment for
a subset of Standard ML. Each function type is represented as a rectangular block
with colored blocks inside that represent argument and result types. A visual
comparison of the blocks can reveal possible conflicting type schemes. Polymorphic
types are represented by multi-striped blocks with each stripe representing a different
type. A scalability issue is that since each type has a representative color, the
number of colors grows linearly with the number of types, and the programmer has
to learn and remember which color is associated with which type.

2.2.2 Types in VPLs

We have already pointed out that few VPLs use explicit type declarations, and
that in the absence of explicit type declarations, language designers are left with the
choice of either dynamic typing or static typing with type inference. To date, most
VPLs (e.g., Prograph [Cox et. al. 1989], KidSim/Cocoa [Cypher and Smith 1995],
Chimera [Kurlander 1993], VIPR [Citrin et al. 1997], and Formulate [Ambler and
Broman 1998]) have chosen dynamic typing.

Interestingly, the disadvantage of dynamic typing’s inability to provide feedback
about type errors until runtime bears re-examination for responsive VPLs. For
responsive VPLs (those languages at liveness level 3 and above on Tanimoto’s
Dynamic type checking can indeed produce immediate feedback about type errors in many cases, due to the fact that at level 3 and above, "run-time," "translation time," and "program-entry time" are intertwined. For example, in spreadsheet languages, which are at level 3, concrete, immediate feedback about type errors can be provided by eagerly evaluating a formula as soon as it is entered, which is even earlier than the feedback about type errors in static approaches for traditional textual languages. If any type error occurs in the course of this evaluation, a special value such as "Error" is displayed in the cell. This approach features simplicity and immediate visual feedback, but unfortunately, it cannot detect all type errors. For example, if cellA had the value "true", the type error in the formula "if cellA then (3+4) else (cellA + 4)" would not be detected.

2.2.2.1 Static types in VPLs

Our search through VPL literature has revealed only seven VPLs that have incorporated static type inference. In about half of these VPLs, systems like Milner's are fully incorporated into the VPL. ESTL [Najork and Golin 1990] and CUBE [Najork 1996] are VPLs in this category. For example, Milner's type system has been incorporated into ESTL as follows. ESTL, an extended version of the dataflow VPL Show and Tell [Kimura et al. 1990], has a feature termed consistency, with which values can be compared, conditions tested, etc. If such conditions are not met, an inconsistency is said to exist. In this case, the inconsistent area is rendered in a different pattern, and processing of affected areas cease to produce output. This

\[ \text{2 At liveness level 1 no semantic feedback is available. At level 2 the user can obtain semantic feedback, but it is not provided automatically (as in compilers and interpreters). At level 3, incremental semantic feedback is automatically provided after each program edit, and all affected on-screen values are automatically redisplayed (as in the automatic recalculation feature of spreadsheets). At level 4, the system responds to edits as in level 3, as well as to other events such as system clock ticks [Tanimoto 1990].} \]
feature originally was developed for Show and Tell as a visual mechanism to replace Booleans. In ESTL, the consistency concept also is used to reflect type validity. The entire type system is visible to the user, including the polymorphic type variables. The types and type variables are represented as icons. Since the type system is a visual rendition of Milner’s type system, the programmer is required to thoroughly understand the Milner system, including polymorphic types, type variables and types of higher order functions.

Clover [Braine and Clack 1996] is a functional and object-oriented VPL. Clover combines traditional object-oriented features such as (single) inheritance, subtyping and method overloading with functional features that include referential transparency, polymorphism, curried partial applications, higher-order functions and lazy evaluation. The language is completely type safe. It, however, places some restrictions on subtypes such as invariant method signatures (subclass method signatures must exactly match the type signatures of the class methods), and it requires explicit declarations of upper bounds on the types of method arguments and results.

A common limitation in many of these VPLs’ type systems is that they do not support user-defined types. Of those systems that do support user-defined types, only the type system of Clover supports inheritance.

2.2.2.2 Understandability of type inference results in VPLs

The remaining VPLs with type inference systems have aimed for greater understandability of type systems, primarily by emphasizing concreteness in the types themselves. Fabrik [Ingalls et al. 1988], which was the first VPL to report the use of type inference, is an example. Fabrik is a dataflow VPL that includes an interactive polymorphic type system with some type inference. Fabrik’s type system is simple, concrete and highly visible. Each node in the dataflow graph contains input and output “pins”. Wires that connect nodes are attached to these pins. Each pin has a type that may be either a primitive type, a compound type constructed from
only primitive types or an unspecified (i.e., polymorphic) type. These types can be declared by the user explicitly, or they can be derived implicitly. Type checking is performed when a user attempts to connect two pins. A pin with an unspecified type acquires a type when it is attached to a pin with a known type. If a type mismatch occurs, a message is displayed, and the connection is not made. This approach to implicit polymorphism seems consistent with the concreteness of the language, but the type system is not as fully developed as that of the other languages discussed here. For example, user-defined types are not handled.

In an unusual application of type inference in VPLs, Pacull introduces a visual type system whose goal is not type safety; rather the system infers and propagates information for rendering purposes [Vion-Dury and Pacull 1997]. The inference system’s primitives are a set of visual items referred to as “basic glyphs”, such as lines, points, polygons and text. These glyphs are defined by tuples of visual attributes such as position, color, size, shape and orientation. The attributes define the way a basic glyph should be rendered on the screen. Complex glyphs are a composite of basic glyphs and acquire their attributes through the inference process. Forms/3’s previous approach to types borrowed heavily from Milner’s approach but was more concrete [Burnett 1993]. The goal was to design a concrete approach to types analogous to “naive physics” where the user sees and experiments with certain concrete entities and draws conclusions about the way things work without proving theorems or dealing with abstract concepts. A significant difference between our previous type system and Milner-like systems is that matrices, user-defined types and primitive types were the only types in Forms/3. No function definition types, tuple types, subtypes, recursive types, union types, higher-order types or type constructors were included. Our previous system was sufficient to handle Forms/3’s features at that time, but it did not have the power to support more advanced features such as inheritance.
2.3 Forms/3

In Chapter 1, the research spreadsheet VPL Forms/3 was briefly introduced. In this section, an informal example of a Forms/3 program is presented, in order to paint a concrete picture of a setting in which the new type system is expected to function. Following the example, a subset of Forms/3 referred to as Core Forms/3 is introduced and formally defined. Core Forms/3 supports the complete semantics of Forms/3. The two main differences between Core Forms/3 and Forms/3 are that the basic formula models are different, and object attributes are simplified in Core Forms/3. Core Forms/3 is used to prototype the new type system because it supports the complete semantics of Forms/3, and its simpler representation results in a small axiom set.

2.3.1 A Forms/3 Example

As we mentioned in Chapter 1, a Forms/3 program consists of one or more forms, each of which includes one or more cells. Cells are created and given formulas through a variety of sequences that consist of direct manipulation and typing. A Forms/3 program is defined by the data relationships between the cells in the program.

In Forms/3, three main kinds of cells can be created: a basic cell, a matrix and an abstraction box. The basic cell is a single object whose value is defined by its formula. The matrix is a collection of cells and regions. Two cells—the NumRows and NumCols cells—define the dimensions of the matrix, that is, they define the number of rows and columns in the matrix. Regions include a subset of the cells that exist in a matrix. For example, a 2 x 2 matrix can be divided into four regions, each of which contains one cell in the matrix. The cells within a region are provided with a subset of a basic cell’s attributes. These cells also have their values defined
by their respective regions' formulas. An abstraction box is a collection of cells or matrices. An abstraction box exists only on a user-defined type form. Each of these types of cells will be formally defined in Section 2.3.2.

Suppose a spreadsheet user such as a population analyst would like to define a visual representation of data using domain-specific visualization rules that make use of the built-in circle type of Figure 2.4. Figure 2.5a presents an example of this visualization in Forms/3. The program categorizes population data into cities, towns and villages. A different sized black circle that is defined by its own copy of the form in Figure 2.4 represents each of these categories.

Figure 2.4. A portion of a Forms/3 form primitiveCircle that defines a circle. The form name is displayed in the window title bar. A user can view and specify formulas by clicking on the formula tabs. Radio buttons and popup menus are the equivalent of cells with simple formulas. The black circle in someCircle is a sample circle (or the user can edit its formula to refer to some other circle); the cells above the horizontal line report on someCircle's attributes. The circle in newCircle is constructed using the specifications in the other cells below the line.

Users can specify formulas textually. For example, the population analyst can define the size of a circle by entering a spreadsheet formula in the conventional
textual manner, such as by typing a formula into the radius cell of Figure 2.4. Alternatively, users can define formulas via direct manipulation and gestures. For example, to define the formula for the cell city, the population analyst can draw the circle gesture in Figure 2.5b. This graphical approach is syntactic sugar for the mechanism of Figure 2.4. That is, this approach defines city’s formula to be a reference to cell newCircle on a copy of the built-in circle definition form whose radius formula is defined to be the radius of the drawn circle gesture. To specify that the circle should be black, the population analyst clicks on the circle to display its definition form and then defines the desired formula for cell fillForeColor (Figure 2.5c). For further information regarding programming in Forms/3 by direct manipulation and gestures, see [Burnett and Gottfried 1998].
Figure 2.5. (a) A visualization of population data. The formula shown for "graph" is shared by the four cells inside the matrix. (The □s in the formula are miniaturized drawings of the cells' current values, which can optionally be included in formula displays.) (b) To define the circle for cell "city", the population analyst first draws a circle gesture in city's formula edit window, and then, (c) after clicking on the resulting circle to display its definition form (in gray because it is a copy; white indicates formulas different from the original), the population analyst specifies the black fillForeColor formula via a popup menu. Each manipulation is immediately reflected textually and graphically in city's formula edit window (shown behind the circle form in (c)).
Figure 2.5 (Continued).
2.3.2 Core Forms/3: The Subset for Formal Reasoning

2.3.2.1 Programming objects and notational conventions

In the previous section, the programming objects in Forms/3 were informally used and described. In this section, we formally define these programming objects in Core Forms/3. All Core Forms/3 programming objects are supported in Forms/3. The cosmetic attributes such as borders and on-screen positions that are found in Forms/3's programming objects, however, are not present in Core Forms/3.

The basic Core Forms/3 programming objects are defined as follows:

A **program** is a set of forms, where each form is identified by a unique formID.

A **form** is a tuple: (formID, modelName, ROset), where ROset is a set of referenceable objects, and

- formID if this form is not a copy or
- modelName = F.modelName if this form is a copy of form F.

The cosmetic attributes such as borders and on-screen positions that are found in Forms/3's programming objects, however, are not present in Core Forms/3.

An **ROset** is a set of referenceable objects, each of which is identified by a unique cellID.

A **referenceable object** (RO) is a cell or a cell group.

A **cell group** is a matrix or an abstraction box.

A **cell** is a tuple: (cellID, ROset, formula, value) whose ROset contains only cells, including one whose cellID is "<MID>[NumRows]" and one whose cellID is "<MID>[NumCols]" where <MID> is the matrix's cellID. (The term gridROset will be used to denote a matrix’s ROset – {<MID>[NumRows], <MID>[NumCols]}.)

A **formula** is defined in Table 2.1.

A form’s modelName is used to track similarities between different forms. An RO is any non-constant that can be referenced in a formula. Unlike the other ROs, a matrix has no value because a matrix is a programming mechanism used to support spreadsheet-like grids of cells. Each of these cells, however, has a value. In this thesis, a matrix’s value occasionally is used as an abbreviation for the values of the cells in its ROset.
As we mentioned before, users have the ability to use built-in types or to create user-defined types. Both built-in types and user-defined types are defined on Visual Abstract Data Type (VADT) forms. VADT forms are type definition forms. VADT forms and their components are defined as follows:

- A **VADT form** is a form whose ROset includes a cell with cellID "Image", one abstraction box with cellID "manias", and zero or more additional ROs.
- An **abstraction box** is a tuple: (cellID, ROset, formula, value) whose ROset contains only cells and matrices. An abstraction box also is an element of a VADT form’s ROset.
- A **virtual RO** is a virtual cell or a virtual matrix.
- A **virtual cell** is a cell whose ROset is empty and is an element of another cell’s ROset.
- A **virtual matrix** is a matrix whose ROset contains only virtual cells and is an element of another cell’s ROset.

VADT forms are used to create user-defined monomorphic types. When a user defines a VADT form with the formID T, a new type named T is defined. Each abstraction box on the VADT form is an instance of the type. The members of an abstraction box’s ROset define the ROs required to create an instance of the type. For example, in Figure 2.6, the abstraction boxes are aPoint and movedPoint. Each abstraction box is an instance of type Point. If a cell Z references the abstraction box aPoint, then Z’s ROset “virtually” corresponds to aPoint’s ROset. That is, Z’s ROset “virtually” contains the x and y cells inside aPoint.
Figure 2.6. The VADT form Point defines type Point and two of its instance, aPoint and movedPoint. Each of the abstraction boxes aPoint and movedPoint is an instance of type Point.

The following notational conventions for programming objects are used:

"Dot notation" specifies elements of a tuple. For example, F.modelName refers to the modelName of F.

← denotes the referencing operation ("ref") when the operand is an RO. For example, X ← Y means that RO X's formula is a reference to RO Y. (The arrow points in the direction of data flow.)

¬ denotes the transitive closure of ←. X ¬ Z iff either X ← Z, or X ← Y and Y ¬ Z.

←c denotes the constant specification operation ("ref") when the operand is a constant. For example, X ←c 5 means that RO X's formula is the constant 5.

∈ denotes the transitive closure of ∈. X ∈ Z iff either X ∈ Z, or X ∈ Y and Y ∈ Z.
2.3.2.2 Formula syntax and semantics

Core Forms/3 consists of only three operators: the implicit operator referencing another RO ("\(\leftarrow\)"), the implicit operator specifying equality to a constant ("\(\leftarrow c\)"), and the explicit operator "compositionOfParts". Each of these operators is defined in Table 2.1.

In Figure 2.5a, the cells in the matrix graph are examples of the first operator because the cells city, town and village are referenced in these cells' formulas. The cells in the matrix population provide examples of the second operator. Each cell's formula is a reference to a constant. In Figure 2.6, the abstraction box aPoint is an example of the third operator. aPoint is composed of the cells x and y.

The syntax given in Table 2.1 includes textual versions of some elements that the user would never actually type in (in either Core Forms/3 or Forms/3). For example, the user never uses the textual defSet syntax. Instead the user clicks on the desired cell, and the system internally records the reference using the defSet notation. The internal notation is presented because it is a useful formal notation to express the type reasoning mechanism.
Table 2.1. The grammar for Core Forms/3's formula language. The divider separates the operator syntax from the ref operand syntax. To minimize the amount of new notation, we also build upon the terms established here in the formal presentation of the model as explained in the text.

| formula | ::= compositionOfParts | expr |
| expr | ::= constant | ref |

| ref | ::= RORef | formRef : RORef |
| formRef | ::= formID | modelName | defSet |
| defSet | ::= ( defSet ) |
| defs | ::= def | def , defs |
| def | ::= RORef = expr |
| RORef | ::= cellID |
| cellID | ::= simpleCellID | matrixID | matrixID [subscripts] |
| | ::= absID | absID [simpleCellID] | absID [matrixID] |
| | ::= absID [matrixID] [subscripts] |
| subscripts | ::= matrixSubscript@matrixSubscript |
| matrixSubscript | ::= expr |

Table 2.2. Axiomatic semantics for each operator in Core Forms/3. (The special provisions for matrices are primarily because matrices do not have values.) Implicit in the notion of equality is the fact that if value1 = value2, then their types are equal. Also note that the "Y &gt;= X" precondition prevents circular references. All the
preconditions are easily checked statically, and in Forms/3 are enforced by the environment.

Based on the definitions provided in the preceding tables, no arbitrary input values exist. All inputs are constants entered into a program, using formula edits. This feature provides us with the ability to apply static type checking to every part of our program [Djang 1998].

### 2.3.2.3 Forms

The three operators described in the previous section can be used to construct new operators that other programming languages often support. Forms/3 provides several built-in forms that represent operators commonly found in other programming languages such as +, -, and if-then-else.

The semantics of each built-in form is defined through preconditions and postconditions. Table 2.3 presents the semantics for the + form. The + form consists of a set of three ROs: plusA, plusB and plusC. The cells plusA and plusB are the two arguments for addition. They are modifiable in order to allow passing in different arguments for different additions needed in a program. The cell plusC contains the result and therefore is not modifiable. Referencing plusC in Core Forms/3 is equivalent to computing “plusA + plusB” in Forms/3.

<table>
<thead>
<tr>
<th>modelName</th>
<th>ROset</th>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>contains modifiable cells with cellIDs plusA, plusB and unmodifiable cell with cellID plusC</td>
<td>plusA.value is a number</td>
<td>plusC.value is a number that is the sum of plusA.value and plusB.value</td>
</tr>
</tbody>
</table>

Table 2.3. Semantics of form + (and of forms copied from +). The term “modifiable” means that the programmer is allowed to edit the formulas for these
cells; “unmodifiable” means the formulas may not be edited. Since no state modification exists in Core Forms/3 (or in Forms/3), the preconditions are invariant. Form + is one of the forms built into Core Forms/3 to provide access to a primitive operation; the others are: -, +, /, mod, =, and, or, >, <, not, width, height, if and compose.

Table 2.4 states the properties of forms. The first property [Copies] states that forms with the same modelName always have ROsets whose cells have the same cellIDs. The + form presented in Table 2.3 is an example. All copies of the + form have the same three cells: plusA, plusB and plusC. The only difference may be the formulas for plusA and plusB. The remaining properties are applicable only to VADT forms.

The properties [VADTformExistsC], [VADTformExistsR] and [Inst] state that for every RO X, there exists a VADT form denoted T_X whose main abstraction box references X. As a result, T:MainAbs.value = X.value, and X.value is type T. Further, each T_X has the same modelName as VADT form T. Hence a constant formula (←_C C) can always be replaced by a reference to T_C:MainAbs. For example, if the constant 3 of type primitiveNumber exists, then there exists a form primitiveNumber_3 = primitiveNumber(MainAbs == 3) whose modelName is primitiveNumber. Similarly, if an RO Z has a Point (15, -7) as its value, then a copy of form Point exists. [Inst] further states that if Z has such a value, it directly or transitively references an abstraction box on the appropriate VADT form (Point, in this example).
Table 2.4. Invariant properties of forms. Entries such as \( \text{T(MainAbs} = C) \) follow the defSet syntax established in Table 2.1. (The subscript notation for set elements is used to represent arbitrary elements in the set and does not imply any position in the set because sets are inherently unordered.)

### 2.3.3 Translating Between Forms/3 and Core Forms/3

In order to translate a Forms/3 programming object to a Core Forms/3 object, the cosmetic attributes are removed from the object. The only situation in which an attribute cannot be removed occurs when an RO is positioned inside another RO. Core Forms/3 interprets this positioning as “compositionOfParts”. For example, if RO X is placed inside RO Y, then X is an element of Y.ROset. In Core Forms/3, Y.formula is the result of one of two cases: if Y is given a formula, then Y.formula is equal to the given formula; otherwise, Y.formula is defined to be “compositionOfParts”. 
After the cosmetic attributes are removed, the RO's formula is translated into a textual Forms/3 formula (Burnett and Gottfried [1998] provide the details for the translation). This formula will contain one or more operators and an appropriate number of operands. Since operands are the same in both systems, only operators need to be translated. In Core Forms/3, the only operator is the reference operator. We therefore translate a Forms/3 operator as a reference to another RO. For example, given the Forms/3 formula for RO X in prefix notation:

\[ X . \text{formula} = \text{op} \ \text{expr1, expr2} \]

the Core Forms/3 formula is a reference to the result RO on the form defining this operation:

\[ X . \text{formula} = \text{op}(\text{arg1} = \text{expr1}, \text{arg2} = \text{expr2}) : \text{Result} \]

where \text{arg1} and \text{arg2} are the cellIDs of modifiable ROs on form \text{op}, and \text{Result} is the cellID of an unmodifiable RO containing the result of the built-in operation. If a Forms/3 formula contains subexpressions, the subexpressions are translated in a bottom-up order. Figure 2.7 presents a simple translation example.

```
\[ \frac{+}{\text{a} = \frac{\ast}{\frac{x}{\text{y}}}} \]
```

Figure 2.7. Translation of the Forms/3 formula \((a + (x \ast y))\) to Core Forms/3 proceeds bottom-up. First, \((x \ast y)\) is translated to \(*(\text{timesA} = x, \text{timesB} = y) : \text{timesC}\). The remaining expression then is translated, and the translated expression in the first step is substituted in. The resulting Core Forms/3 formula is: \(+ (\text{plusA} = a, \text{plusB} = *(\text{timesA} = x, \text{timesB} = y) : \text{timesC}) : \text{plusC}\).

In translating Core Forms/3 to Forms/3, only one additional mechanism is needed because the remaining Core Forms/3 operators are a subset of Forms/3. The operator "compositionOfParts" needs to be handled by positioning all members of an
RO's ROset within the RO's borders. For example, if RO X is a member of RO Y's ROset, then X must be positioned inside Y's borders. Furthermore, since Y.formula is defined to be "compositionOfParts", RO Y will not have a formula, and X's formula is defined by X.formula.
Chapter 3: Djang et al.'s Model of Types—Related Work

Djang et al. [1998] developed a new model of types that incorporated implicit static typing into a declarative VPL. This new model of types also supported the various forms of inheritance without the introduction of explicit type declarations. Djang et al.'s model of types guarantees that a program is type-safe and reduces the programming complexities associated with types and type declarations. This chapter reviews Djang et al.'s model in depth.

The first section introduces the Djang et al.'s model of types. The axioms for the type inference and type checking procedures are reviewed in the following sections. A type inference example that involves no inheritance is presented in Section 3.5. The axioms for the type inference that supports inheritance are introduced in the following section. The chapter concludes with examples of type inference and type checking in the presence of inheritance.

3.1 Model of Types: Fine-grained Reasoning in Terms of Guarantees versus Requirements

The goals of Djang et al.'s research were to develop a model of types comprehensive enough to be usable in VPLs for programmers with powerful features such as polymorphism and fine-grained inheritance while at the same time being potentially understandable enough for use even in VPLs intended for end users. In order to achieve the first goal without sacrificing the second, Djang et al.'s type model reasons at a fine-grained level about individual operations guaranteed and required rather than about entire types as atomic units. Because all reasoning is done in terms of guarantees and requirements, subtypes, complex compositions of types and interfaces do not exist.
At this granularity of reasoning, the need to reintroduce declarations of interfaces or subtype relationships is eliminated. Reintroduction of type declarations conflicts with the goal of potential use by end users. For example, if cell X is being added to something, then a requirement exists that X support number operations. If cell X references a number constant, then it guarantees number operations. If all the requirements for an RO are met by the RO's guarantees, the RO is type safe. That is, if the requirements are not a subset of the guarantees, then a type error exists. In Djang et al.'s type system, the guarantees and requirements are statically inferred.

More formally, if the set of operations that are inferred to be guaranteed for RO X are denoted G(X), and the set of operations inferred to be required of RO X are denoted R(X), then Djang et al.'s model of types defines type safety as follows:

Definition: If \( \forall \text{ROs } X \in \text{program } P, \text{R}(X) \subseteq G(X) \), then P is type safe.

### 3.2 Guarantee Sets

In general, each RO guarantees all the operations defined on the VADT form corresponding to its value (this form was defined by Table 2.4's [VADTformExistsR]). In Core Forms/3, each operation is associated with an RO and is synonymous with a cellID.

The simplest kind of guarantee set to infer is that for an abstraction box. Abstraction boxes reside only on VADT forms which thus identify their type. Thus the guarantee set for an abstraction box F:A is the collection of operations available on the VADT form F. Since operations are cellIDs, the axiom is:

\[
\text{G(F:A) = } \{ x \mid x \in F.\text{ROset} \}\text{ where A is an abstraction box}
\]

This axiom is applicable to both user-defined types and built-in types. For example, G(primitiveCircle:newCircle) includes all the ROs on form primitiveCircle (see Figure 2.4): the operations radius, thickness, lineStyle, lineForeColor, etc., including the abstraction boxes someCircle and newCircle. Other circle-related tasks that can be constructed using these low-level operations do not need to be included...
on the primitiveCircle form itself. In this thesis, we abbreviate the set of low-level operations for these primitive types (the ROs on their VADT forms) as "<primitiveType>Operations". In this example, G(primitiveCircle:newCircle) = "primitiveCircleOperations".

Axiom [GC] says that ROs with constant formulas simply derive their guarantee sets from the primitive form describing the constant’s value (see Table 3.1):

\[
\text{[GC]} \quad X \leftarrow_c C \quad \text{where } C \text{ is a constant} \\
G(X) = G(C) \text{ where } G(C) = \{ y \mid y \in F_C.\text{ROset} \}
\]

<table>
<thead>
<tr>
<th>Example constants C</th>
<th>G(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = 3</td>
<td>primitiveNumberOperations</td>
</tr>
<tr>
<td>C = &quot;hello&quot;</td>
<td>primitiveTextOperations</td>
</tr>
<tr>
<td>C = true</td>
<td>primitiveBooleanOperations</td>
</tr>
</tbody>
</table>

Table 3.1. Every constant value guarantees exactly the operations on its primitive form. For succinctness, we abbreviate these sets <primitiveType>Operations rather than listing the individual ROs.

For cells and matrices that reference other cells and matrices, the guarantee set simply propagates from the referenced cell or matrix by axiom [Gref] below. The "where" clause restricts this axiom to cells and matrices because it is not needed for abstraction boxes; they are already handled by axiom [GA]. Removing this restriction, however, would not cause any adverse effects because its removal would not introduce any conflicts with [GA].

\[
\text{[Gref]} \quad X \leftarrow Y \quad \text{where } X \text{ is a cell or matrix} \\
G(X) = G(Y)
\]

The previous three axioms handle every legal Core Forms/3 formula except matrices with compositionOfParts formulas. In this case, axiom [GM] says that the guarantee set is derived from the guarantees of the matrix’s ROset. This axiom
could have the precondition "M.formula = compositionOfParts," but it is not necessary because the resulting guarantee set will be the same regardless of whether [Gref] or [GM] is applied to a matrix with a reference ("←") formula.

\[ [GM] \quad G(M) = \bigcap G(M[i]) \quad \text{where } M[i] \in M.\text{gridROset} \]

[GM] is one of several elements in this model that are different for matrices than for other ROs. Djang et al. could have changed the language definition of matrices to make a matrix define a complex value, in order to eliminate most of the specialized matrix reasoning. The authors elected not to do so because reasoning about Core Forms/3 matrices demonstrates how the model can be applied to other VPLs’ groups of objects (such as grids in spreadsheets and in rule-based demonstrational systems) that do not produce a single value.

Finally, regarding primitive operations, guarantees are provided for ROs on the primitive forms via the postconditions that define their semantics. For example, the result cell (plusC) of form + guarantees all the operations guaranteed for built-in type number (which are enumerated via axiom [GA] and [GC]). Postcondition guarantees for some of the primitive forms are given in Table 3.2.
<table>
<thead>
<tr>
<th>modelName</th>
<th>ROset</th>
<th>Type-related postconditions: Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC.</td>
<td>G(plusC) = primitiveNumberOperations</td>
</tr>
<tr>
<td>&gt;</td>
<td>contains modifiable cells with cellIDs greaterThanA, greaterThanB, and unmodifiable cell with cellID greaterThanC.</td>
<td>G(greaterThanC) = primitiveBooleanOperations</td>
</tr>
<tr>
<td>append</td>
<td>contains modifiable matrices with cellIDs appA, appB, and unmodifiable matrix with cellID appC.</td>
<td>G(appC) = G(appA) $\cap$ G(appB)</td>
</tr>
<tr>
<td>if</td>
<td>contains modifiable cells with cellIDs ifA, ifB, ifC, and unmodifiable cell with cellID ifD.</td>
<td>G(ifD) = G(ifB) $\cap$ G(ifC)</td>
</tr>
</tbody>
</table>

Table 3.2. Guarantee sets for some primitive forms. Type postconditions are stated as guarantees under our model of types; this table shows these postconditions for a few of the primitive forms. (Compare these with the previous type-related postconditions of Table 2.3.) Preconditions for these forms will be given in the next subsection as requirements.

3.3 Requirement Sets

Guarantees normally propagate with dataflow and in Core Forms/3 usually have only one source (since a Core Forms/3 formula has no more than one reference in its formula). Requirements, however, propagate against dataflow because they indicate how the RO is to be used by other parts of the program. Further, since many parts of the program may make use of (reference) a single RO, all of these uses must be collected. The implication of these differences is that requirement sets must be aggregated and propagated via set unions, rather than via the equality assertions that sufficed for most of the guarantee sets.
From the previous paragraph, the requirements axiom appears to be defined by the union of all the requirements of ROs referencing it, as in:

\[ R(Y) = \bigcup_{i=1..n} R(X_i) \]

where \( X_i \) is not an abstraction box

Abstraction boxes referencing \( Y \) are already "safe" from type errors because their definitions and properties prevent impersonations. Hence, it is not necessary to propagate requirements from abstraction boxes that reference \( Y \) into the requirements of \( Y \), and such references can simply be ignored in the requirements axiom.

The above version is adequate for propagating requirements that are already present in the system, but it does not allow new requirements to enter the system. The primary way for new requirements to be initiated is for an operation on a VADT form to be referenced. That is, \( R(Y) \) needs to include the operations in \( F_Y \cdot \text{ROset} \) that are actually being used (referenced) as well as the operations given in the axiom version above.

\[ R(Y) = \bigcup_{i=1..n} R(X_i) \]

where \( X_i \) is not an abstraction box.

This version, however, overlooks the fact that copies of \( F_Y \) (multiple VADT forms whose main abstraction boxes reference \( Y \)) may have individual differences in some of the formulas of their other ROs. For example, there could be two copies of form Point (from Figure 2.6) with \( \text{aPoint} \) on both copies referencing cell \( Y \), but with cell \( \text{delta-x} \) on one of the copies referencing some cell \( V \), and with cell \( \text{delta-y} \) on the other copy referencing some cell \( W \). In defSet notation, the first copy is \( \text{Point} (\text{aPoint} \equiv Y, \text{delta-x} \equiv V) \) and the second copy is \( \text{Point} (\text{aPoint} \equiv Y, \text{delta-y} \equiv W) \). Thus, additional requirements for \( Y \) are that its type definition form also
includes operations that appear in these defSets, namely aPoint, delta-x, and delta-y in this example. Changing the second precondition to defSet notation and adding a third union factor to the conclusion leads to the final version below:

\[
\begin{align*}
X_1, X_2, \ldots, X_n &\leftarrow Y \quad \text{and} \quad Z_1 \leftarrow F_r(\text{defSet}_1); O_{p1}, Z_2 \leftarrow F_r(\text{defSet}_2); O_{p2}, \ldots, \\
Z_m &\leftarrow F_r(\text{defSet}_m); O_{pm}
\end{align*}
\]

\[
\begin{align*}
R(Y) &= \bigcup_{i=1}^{n} R(X_i) \quad \cup \{O_{p1}, O_{p2}, \ldots, O_{pm}\} \cup \{\text{Op}\mid \text{Op} \in \text{arg}_k\}
\end{align*}
\]

where \(\text{arg}_k = \{\text{arg}\mid \text{arg} = \text{expr} \in \text{defSet}_k\}\) and where \(X_i\) is not an abstraction box.

In summary, what this final version of \([R1]\) says is that the requirements of \(Y\) include the requirements of ROs referencing \(Y\) (these are the \(X_i\)s in the axiom), the operations on all copies of \(Y\)'s VADT form that any RO is actually referencing (the \(O_{pi}\)s referenced by the \(Z_i\)s) and any additional operations on all copies of \(Y\)'s VADT form that are in the defSets defining the form copy (the last \(O_{pm}\) set in the axiom's conclusion).

A matrix's NumRows and NumCols cells are always known to require primitiveNumberOperations:

\[
[RN] \quad R(N) = \text{primitiveNumberOperations}
\]

where \(N \in M.\text{ROset}\) and either \(N.\text{ID} = M.\text{ID}[\text{NumRows}]\) or \(N.\text{ID} = M.\text{ID}[\text{NumCols}]\).

Matrices require everything that the cells in their gridROsets require:

\[
[RM] \quad R(M) = \bigcup R(M[i]) \quad \text{where} \ M[i] \in M.\text{gridROset}
\]

Preconditions on primitive forms' ROs add to the requirements propagating through the system (see Table 3.3). For example, in the \(+\) form, the ROs plusA and plusB now require primitiveNumberOperations in addition to the postconditions previously given for form \(+\).
<table>
<thead>
<tr>
<th>ModelName</th>
<th>ROset</th>
<th>Type-related preconditions: Requirements</th>
</tr>
</thead>
</table>
| +         | contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC. | R(plusA) = primitiveNumberOperations  
R(plusB) = primitiveNumberOperations |
| >         | contains modifiable cells with cellIDs greaterThanA, greaterThanB, and unmodifiable cell with cellID greaterThanC. | R(greaterThanA) = primitiveNumberOperations  
R(greaterThanB) = primitiveNumberOperations |
| append    | contains modifiable matrices with cellIDs appA, appB, and unmodifiable matrix with cellID appC. | if tempR = application of [R1] to appA, then R(appA) = tempR ∪ R(appC)³  
if tempR = application of [R1] to appB, then R(appB) = tempR ∪ R(appC) |
| if        | contains modifiable cells with cellIDs ifA, ifB, ifC, and unmodifiable cell with cellID ifD. | R(ifA) = primitiveBooleanOperations  
if tempR = application of [R1] to ifB, then R(ifB) = tempR ∪ R(ifD)  
if tempR = application of [R1] to ifC, then R(ifC) = tempR ∪ R(ifD) |

Table 3.3. Requirements sets for some primitive forms. For the primitive forms, type preconditions are stated as requirements under our model of types; this table shows these preconditions for a few of the primitive forms. (The preconditions are invariant, but to avoid clutter, we did not explicitly repeat them in the postconditions of Table 3.2.)

³Or, more formally,

\[
X_1, X_2, \ldots, X_n \leftarrow \text{appA and } Z_1 \leftarrow F_{\text{appA(defSet_1):Op_1}}, Z_2 \leftarrow F_{\text{appA(defSet_2):Op_2}}, \ldots, Z_m \leftarrow F_{\text{appA(defSet_m):Op_m}}
\]

\[
R(\text{appA}) = \cup_{i=1..n} R(X_i) \cup \{Op_1, Op_2, \ldots, Op_m\} \cup R(\text{appC})
\]
3.4 Recursion

Most "function calls" (uses of new copies of forms) can be statically type checked when a "call" (reference) to an RO on one of these copies is edited into a formula. This static strategy, however, cannot be employed successfully with recursion because the number of form copies that will be generated by the recursive call cannot be determined statically. To solve this problem, the next two axioms provide a conservative static determination of the guarantee set of an RO involved in recursion; they say that if one branch of an "if" is recursive, then it only guarantees what the non-recursive branch guarantees.

[RecB]

\[
\begin{align*}
F:X & \leftarrow \text{if}(\text{ifA} \equiv \text{Adef}, \text{ifB} \equiv \text{Bdef}, \text{ifC} \equiv \text{Cdef}) : \text{ifD}, \\
& \quad \text{there is a recursive reference in Bdef}^b \\
G(F:X) &= G(\text{if}(\text{ifA} \equiv \text{Adef}, \text{ifB} \equiv \text{Bdef}, \text{ifC} \equiv \text{Cdef}) : \text{ifC})
\end{align*}
\]

[RecC]

\[
\begin{align*}
F:X & \leftarrow \text{if}(\text{ifA} \equiv \text{Adef}, \text{ifB} \equiv \text{Bdef}, \text{ifC} \equiv \text{Cdef}) : \text{ifD}, \\
& \quad \text{there is a recursive reference in Cdef} \\
G(F:X) &= G(\text{if}(\text{ifA} \equiv \text{Adef}, \text{ifB} \equiv \text{Bdef}, \text{ifC} \equiv \text{Cdef}) : \text{ifB})
\end{align*}
\]

3.5 Example: Type Inference Without Inheritance

The population example in Figure 2.5 is a program with no inheritance. Each different circle represents a different city population. For this type inference example, we use full Forms/3 because it is a real environment, and screenshots of example programs are available. Although the axiom set is given for Core Forms/3,

---

4 Some type inference systems require a special construct to statically detect recursion. In Core Forms/3, no such device is needed. A recursive call exists in Bdef if either Bdef references F':X, or if a recursive call exists in \{recdef₁ | Bdef = someForm(arg₁=recdef₁, arg₁=recdef₂,...):someCell\}, where F.modelName = F'.modelName.
formal reasoning about full Forms/3 is possible using the translations between Forms/3 and Core Forms/3 specified in Section 2.3. (For conciseness, we omit form names where doing so does not introduce ambiguity.)

Notice that in the case of ROs whose only purpose is to display answers on the screen, the requirement set will always be empty. The matrix element population:location[1@1], which contains the value "Portland", is an example of such a cell because no other RO references it:

$$R(\text{location}[1@1]) = \{\} \quad [\text{R1}]$$

$$G(\text{location}[1@1]) = G(\text{"Portland"}) = \text{primitiveTextOperations} \quad [\text{GC}]$$

Obviously, $$R(\text{location}[1@1]) \subseteq G(\text{location}[1@1])$$, so no type error exists here. The same axioms apply to the other cells in location’s gridROset with exactly the same results.

The city cell is another example of an RO with an empty requirement set, but the derivation is a little lengthier because city is referenced by other ROs in the program. For clarity of reasoning, we translate the formula for the cells in the graph matrix’s gridROset to the Core Forms/3 equivalent. For example, a copy of the if form exists on which ifB $\leftarrow$ population:city. In the context of that if form copy, the reasoning proceeds as follows:

$$R(\text{city}) = R(\text{ifB}) \text{ where ifB} \leftarrow \text{population:city on a copy of the}$$

$$\text{primitive if form} \quad [\text{R1}]$$

$$= R(\text{ifD}) \text{ where ifD is on the same copy of that if form} \quad [\text{Table 7}]$$

$$= R(\text{graph}[1@1]) \cup R(\text{graph}[2@1]) \cup$$

$$R(\text{graph}[3@1]) \cup R(\text{graph}[4@1]) \quad [\text{R1}]$$

$$= \{\} \quad [\text{R1}]$$

$$G(\text{city}) = G(290-\text{primitiveCircle:newCircle}) = \text{primitiveCircleOperations} \quad [\text{GC}]$$

Thus, $$R(\text{city}) = \{\} \subseteq \text{primitiveCircleOperations} = G(\text{city})$$

The town and village cells have similar derivations and results.

Cell graph[1@1]’s formula is a reference rather than a constant. Again translating the formula for the cells in the graph matrix to the Core Forms/3
equivalent, let \( \text{if1} \) and \( \text{if2} \) be the appropriate copies of "if". For example, two
translated formulas are: \( \text{if2}: \text{C} \leftarrow \text{village} \) and \( \text{if1}: \text{C} \leftarrow \text{if2}: \text{id} \).

\[
\text{R}(\text{graph}[1 @ 1]) = \emptyset \quad \text{[R1]}
\]
\[
\text{G}(\text{graph}[1 @ 1]) = \text{G}(\text{if1}: \text{id})
= \text{G}(\text{city}) \cap \text{G}(\text{if2}: \text{id})
= \text{G}(\text{city}) \cap (\text{G}(\text{town}) \cap \text{G}(\text{village}))
= \text{primitiveCircleOperations} \cap (\text{primitiveCircleOperations} \cap \\
\text{primitiveCircleOperations})
= \text{primitiveCircleOperations} \quad \text{[GC]}
\]
\[
\text{R}(\text{graph}[1 @ 1]) = \emptyset \quad \text{[R1]}
\]

Thus, \( \text{R}(\text{graph}[1 @ 1]) = \emptyset \subseteq \text{primitiveCircleOperations} = \text{G}(\text{graph}[1 @ 1]) \)

Similar derivations and results apply for the other cells in graph’s gridROset.

Turning to a cell with a non-empty requirement set, the \( \text{population}[1 @ 1] \) cell
has a requirement set of \text{primitiveNumberOperations}. (Forms ">1" and ">2" are
copies of form ">" on which \text{greaterThanA} \leftarrow \text{population}[1 @ 1].)

\[
\text{R}(\text{population}[1 @ 1]) = \text{R}(>1: \text{greaterThanA}) \cup \text{R}(>2: \text{greaterThanA}) \quad \text{[R1]}
= \text{primitiveNumberOperations} \quad \text{[Table 7]}
\]
\[
\text{G}(\text{population}[1 @ 1]) = \text{G}(450000) \quad \text{[GC]}
= \text{primitiveNumberOperations}
\]

Thus, \( \text{R}(\text{population}[1 @ 1]) = \text{primitiveNumberOperations} \subseteq \\
\text{primitiveNumberOperations} = \text{G}(\text{population}[1 @ 1]) \)

Similar derivations and results apply to the other cells in population’s
gridROset. The matrix population is an example of a Forms/3 matrix with the
implicit “compositionOfParts” operator which translates to Core Forms/3’s explicit
use of that operator. Hence, its guarantee and requirement sets are derived solely
from the cells in its gridROset.

\[
\text{R}(\text{population}) = \text{R}(\text{population}[1 @ 1]) \cup \text{R}(\text{population}[2 @ 1]) \\
\cup \text{R}(\text{population}[3 @ 1]) \cup \text{R}(\text{population}[4 @ 1]) \quad \text{[RM]}
= \text{R}(>\text{greaterThanA}) \cup \text{R}(23->: \text{greaterThanA}) \quad \text{[R1]}
= \text{primitiveNumberOperations} \quad \text{[Table 7]}
\]
\[
\text{G}(\text{population}) = \text{G}(\text{population}[1 @ 1]) \cap \text{G}(\text{population}[2 @ 1]) \\
\cap \text{G}(\text{population}[3 @ 1]) \cap \text{G}(\text{population}[4 @ 1]) \quad \text{[GM]}
\]
Thus, \( R(\text{population}) = \text{primitiveNumberOperations} \subseteq \text{primitiveNumberOperations} = G(\text{population}) \)

In the same way, the matrix location can be shown to have an empty requirement set and a guarantee set of \text{primitiveTextOperations} and the matrix graph can be shown to have an empty requirement set and a guarantee set of \text{primitiveCircleOperations}.

Each of the three matrices on the form \text{population} also has a NumRows cell and a NumCols cell in its ROset. They all have the same requirement and guarantee sets, so we provide only one example here.

\[
\begin{align*}
R(\text{location}[\text{NumRows}]) &= \text{primitiveNumberOperations} \quad \text{[RN]} \\
G(\text{location}[\text{NumRows}]) &= G(4) = \text{primitiveNumberOperations} \quad \text{[GC]}
\end{align*}
\]

Thus, \( R(\text{location}[\text{NumRows}]) = \text{primitiveNumberOperations} \subseteq \text{primitiveNumberOperations} = G(\text{location}[\text{NumRows}]) \)

Since every RO on the population form satisfies the constraint that its requirement set be a subset of its guarantee set, the program \text{population} is type safe.

Now suppose a new cell \( X \) is added to form \text{population}, and \( X \)’s formula is “\( \text{population:location}[1 @ 1] + 5 \)”. A type error will occur because the matrix cell \( \text{location}[1 @ 1] \) guarantees \text{primitiveTextOperations} whereas it is required to support \text{primitiveNumberOperations}. More precisely, since \( R(\text{location}[1 @ 1]) = R(+5:\text{plusA}) = \text{primitiveNumberOperations} \), where \(+5\) is the appropriate copy of + for adding 5, and since \( G(\text{location}[1 @ 1]) = \text{primitiveTextOperations} \), then \( R(\text{location}[1 @ 1]) \nsubseteq G(\text{location}[1 @ 1]) \), and hence the program is not type safe.
3.6 Type Inference With Similarity Inheritance

To add similarity inheritance to the basic axioms already presented, it suffices to change only the guarantee axiom [GA]:

\[ [GA'] \quad G(F:A) = \{ x, "like" y \mid x \in F:ROset, y \rightarrow x \} \]

where \( A \) is an abstraction box

This new version says that the guarantee set includes not only every operation defined on form \( F \) but also the word "like" prepended to every operation from which the operations on form \( F \) are inherited. For example, since in Figure 3.1 and Figure 3.2, \( \text{Stack} \rightarrow \text{Queue} \). \( G(\text{Queue}:\text{Queue}) \) then includes not only the operations on form \( \text{Queue} \), but also the word "like" with the \( \text{Stack} \) operation \( \text{top} \), the word "like" with the \( \text{Stack} \) operation \( \text{pop} \), and so on. Inherited operations are already included on form \( F \) (recall the self sufficiency property), but the user is allowed to rename them, such as by changing the name of inherited cell \( \text{top} \) to \( \text{front} \) on the \( \text{Queue} \) form, so by explicitly including the "like" top' entry, the operation is known to the inference system by all of its aliases.
Figure 3.1. The Stack form.
A revised definition of type safety is now needed that takes into account the "like" entries in guarantee sets:

Revised Definition: If $\forall$ ROs $X \in$ program $P$ and $\forall$ Op $\in$ R(X) $\Rightarrow$
- either Op $\in$ G(X) or "like" Op $\in$ G(X), then $P$ is type safe.

This definition says that every required operation needs to either be present in the guarantee set, or the operation from which it is inherited, prepended with the word "like", needs to be present in the guarantee set. As a corollary to this revised definition, a type error is now defined as: $\exists$ X, Op such that Op $\in$ R(X), Op $\notin$ G(X), and "like" Op $\notin$ G(X).
3.7 Example: Type Inference in the Presence of Single Inheritance

The Stack and Queue examples in Figures 3.1 and 3.2 show how Djang et al.'s model of types works in the presence of single inheritance. First we consider the guarantee set of each form's main abstraction box. The guarantee sets for these abstraction boxes are lengthy because they include every RO ∈ the ROSet for the forms.

\[
G(\text{Stack:Stack}) = \{ \text{Stack, push, pop, top, Image, new, lines, new-matrix, Stack[items], Stack[items][NumRows], Stack[items][1@1],...} \} \quad \text{[GA']}
\]

\[
G(\text{Queue:Queue}) = \{ \text{Queue, enqueue, dequeue, front, Image, new, lines, new-matrix, Queue[items], Queue[items][NumRows], Queue[items][1@1],..., "like" Stack, "like" pop, "like" top, "like" Stack[items], "like" Stack[items][NumRows],...} \} \quad \text{[GA']}
\]

Since form Queue was created via similarity inheritance from form Stack, the guarantee sets for abstraction boxes on a Queue form include several operations inherited from Stack ("like" pop, "like" top, etc.) Notice, however, that these guarantee sets do not include "like" push', because the programmer overrode the similarity between Stack's push and Queue's enqueue.

Figure 3.3 shows a simple form with some uses of operations on a stack. Cell collection references a Stack, so its guarantee set is the same as G(\text{Stack:Stack}). Assuming that cells removed-item and the-rest are the only references to cells that perform operations on cell collection, cell collection's requirements set contains two operations:

\[
R(\text{collection}) = \{ \text{top, pop} \} \quad \text{[R1]}
\]

Since G(\text{Queue}) on every copy of form Queue includes "like" top' and "like" pop', a reference to Queue on one of these copies by cell collection would have triggered generalization of the-rest and removed-item, thereby preserving type safety according to the revised definition of type safety in Section 3.6.
Figure 3.3. Cells removed-item and the-rest contains examples of references that will eventually become polymorphic. The figure shows the formulas before generalization. Cell collection’s formula (not shown) references some instance of a Stack, and form 348-Stack’s main abstraction box, Stack, in turn references cell collection. If cell collection’s formula is changed to reference, for example, a Queue, the formulas of removed-item and the-rest will be generalized.

3.8 Example: Type Inference in the Presence of Multiple Inheritance

The Deque (double-ended queue) in Figure 3.4 is an example of multiple inheritance. Deque inherits most of its ROset from Queue, but it also inherits the push operation from Stack. Deque’s main abstraction box’s guarantee set is not much different from that of Queue’s.

\[ G(Deque:Deque) = \{ Deque, enqueue, dequeue, front, Image, new, lines, new-matrix, Deque[items], Deque[items][NumRows], Deque[items][1@1], ..., "like" Stack, "like" pop, "like" top, "like" Stack[items], "like" Stack[items][NumRows], ..., push, "like" Queue, "like" Queue[items], "like" Queue[items][NumRows], ... \} \]  

[GA']

Due to the fine-grained granularity of our model, the presence of multiple inheritance in a program does not significantly affect the derivations of guarantee and requirement sets of operations. The same axioms are applied regardless of the presence and the form of inheritance.
Figure 3.4. A Deque form inheriting from the forms and operations for Stack and Queue.
Chapter 4: Design, Algorithms and Complexity

In this chapter, we present an implementation of the theoretical model presented in Chapter 3. The type system is implemented in the research VPL Forms/3. The first section introduces the data structures used to represent the sets of guaranteed and required operations in Djang et al.’s type model. In the second section, the algorithms for the type inference and type checking mechanisms are formally described. The third section presents the space and time analyses of the data structures and algorithms.

4.1 Data Structures

Forms/3 is implemented in Lucid Common Lisp version 5. The RO data structures are represented using Lisp’s Common Lisp Object System (CLOS). CLOS uses generic functions, multiple inheritance and declarative method combinations to provide an object-oriented extension to Common Lisp [Steele 1990]. The main programming objects in CLOS include classes, generic functions and methods. Forms/3’s ROs are implemented using classes. In Common Lisp, a class is defined by a set of data values and a set of behaviors. The data values can be defined using slots. A slot specifier consists of a slot name and a value [Steele 1990]. Figure 4.1 shows the RO hierarchy in Forms/3. Child classes inherit their parent classes’ respective slots and methods. In order to make use of inheritance, the functions for Djang et al.’s type system were written as general functions or methods using the RO hierarchy. In our implementation, Core Forms/3 assumptions were used, in order to conform to the axioms provided by Djang et al. [1998].
The four main data structures discussed in this section include: operations, sets of operations, primitive type operations and primitive form operations. The complexity aspects of these data structures will be presented in Section 4.4.

4.1.1 Representation of Operations

In our type system, the ROs in a VADT form's ROset are considered operations. Operations are represented by their respective names. If an operation is not given a name, its cellID is used as its cell name. When an operation is created, it does not have a name, so its operation name defaults to its cellID. (Although an RO's operation name may be the same as its cellID, these two values are not equivalent because cellIDS cannot be changed over the course of an RO's existence, whereas an RO's name can be changed by the user.)

Operation names were selected to represent operations because the usage of operation names results in type checking that is sound with respect to type safety. Soundness is ensured: if an operation is in an RO's set of guaranteed operations, then the operation can be used successfully (see Djang et al. [1998] for the proof).
The other options for representing operations included cellIDs and combinations of cellIDs or operation names with formIDs. CellIDs were not selected to represent operations because using this representation would result in a type system that is not sound. The type system would not be sound because an operation may be in a set of guaranteed operations that cannot be used successfully. The reason for this unsoundness is that cellIDs cannot be altered. For example, in Figure 3.2, the form Queue is a copy of the form Stack. If an RO X references Queue:Queue, it guarantees all the operations on the form Queue. Using cellIDs, Queue:enqueue and Stack:push would have identical cellIDs in Forms/3. G(X) then would not be reliable because it would be interpreted by the system to guarantee Queue:enqueue and Stack:push. This evaluation is unsound because the two operations are not equivalent—Stack:push cannot be used successfully. The combination of a cellID or an operation name with the operation’s formID would result in a type system that is too restrictive. For example, the operation Queue:new is inherited from Stack:new (see Figure 3.2). Using formID in the operation’s name would lead the type system to infer that an RO X that references Queue:Queue guarantees Queue:new but not Stack:new. In our example, however, Queue:new and Stack:new are equivalent.

4.1.2 Representation of Sets of Operations

In the previous section, we established that each operation is represented by its name, where a missing name defaults to the operation’s cellID. A set of guaranteed operations is represented as a hash table of operation names, keyed by operation name. The entry for the key is the operation name. No additional information is necessary. A set of required operations also is represented as a hash table of operation names. The key is the operation name. The entry, however, is an integer counter. The counter is used to maintain the number of occurrences of the operation when union is performed on sets of required operations.

The hash table was selected to represent a set of operations because it provides the fastest expected search time, and it is a programming object supported by
Common Lisp. For hash tables, search time is $O(n)$. The worst case, however, arises only when all elements hash to the same location. Cormen, Lieserson and Rivest [1996] write: “Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$.” In our analyses, we assume that the search time is $O(1)$ because operations on a VADT form have unique names, and the probability of operations hashing to the same location is reduced. A data structure that supports a fast search time is desirable because our type inference and type checking algorithms are composed mainly of search functions. For example, the algorithm for “is $R(X) \subset G(X)$?” involves searching in $G(X)$ for each element in $R(X)$. The other data structures we considered were characterized by slower search times. For example, another Lisp supported data structure is the list data structure. For lists, both worst case and expected search time is $O(n)$, where $n$ is the number of elements on the list. Vectors (one-dimensional arrays) and arrays also result in $O(n)$ search time. The second reason for selecting hash tables is that the hash table is a data structure supported by Common Lisp. Other data structures not supported also have slower search times. For example, binary search trees have search times of $O(\log n)$.

Each RO has a slot named “guaranteedOps”. This slot contains a pointer to a hash table, representing the RO’s set of guaranteed operations. The contents of the entry are irrelevant because the contents are never used in the type inference and type checking algorithms. Only the key is used to determine whether or not an operation is a member of a set of operations. A feature of the set of guaranteed operations is that ROs with the same set of guaranteed operations can share the same hash table. That is, each RO’s slot guaranteedOps can contain a pointer to the same hash table. When a new operation is added to a VADT form, its name is added to the main abstraction box’s set of guaranteed operations. This information then is shared by all affected ROs (except for matrices which require their guarantees to be re-inferred), eliminating the need to propagate information. Similarly, when an operation is removed from a VADT form, its name is removed from all affected sets of guaranteed operations.
Each RO also has a slot named "requiredOps". The data structure representing the set of required operations, however, differs from the data structure for guaranteed operations in two respects. First, the entry for each operation name consists of an integer counter. This counter is used to reduce the time to union two or more sets of operations. Second, a set of required operations cannot be shared between two or more ROs.

4.1.3 Representation of Primitive Type Operations

Since primitive type operations cannot be edited with respect to the addition, removal or the renaming of predefined operations, the guaranteed operations for all instances of a given primitive type are composed of the same operations. Information regarding these operations can be stored at system start-up time. For primitive type operations, each type's set of operations is placed in a hash table, keyed by operation name. The entry only consists of the operation name. Each type's hash table then is stored in a global hash table, keyed by primitive type name. In our implementation, this global hash table is referred to as $PrimitiveTypeOperations.

4.1.4 Representation of Primitive Form Operations

The ROs on primitive forms such as + and * have predefined sets of required and guaranteed operations (see Tables 3.2 and 3.3). These preconditions and postconditions are saved in global hash tables referred to as $PrimitiveFormsG and $PrimitiveFormsR, where G and R respectively reflect the sets of guaranteed and required operations. Since the ROs on these primitive forms consist of both requirements and guarantees, the data structures presented in Section 4.1.2 are used to store these operations.
4.2 Algorithms

In our implementation of Djang et al.'s model of types, the four main categories of algorithms include: type inference, adding/removing operations, renaming operations and type checking. These algorithms are derived from the axioms provided in Djang et al. [1998] as well as the data structures developed in the previous section. Time analyses of these algorithms will be presented in Section 4.4.

4.2.1 Type Inference

Type inference is used in our system because the Djang et al.'s model of types supports implicit static typing. As we mentioned earlier, implicit typing relieves the programmer of the additional programming mechanisms associated with type declarations. Type inference is used to determine the sets of guaranteed and required operations for an RO X and all affected ROs when X's formula is edited. Type inference also is used to propagate the guarantees and requirements throughout the system. After each formula edit, type inference is performed because our model supports incremental static typing. That is, our system will detect any and all type errors after each formula edit.

When an RO X's formula is edited, three levels of type inference are performed. First, X's set of guaranteed operations is inferred from its new formula. X's set of required operations is not affected because no new ROs reference it, and no references have been removed. Second, the requirements must be inferred for affecting ROs Y and Z. The RO Y previously referenced by X has its requirement set un-unioned from R(X). All other affecting ROs Y also are updated. The RO Z now referenced by X has its requirement set unioned with R(X). Third, guarantees must be inferred for affected ROs. All ROs that reference X are affected with respect to their sets of guaranteed operations. This section is divided according to these three levels of type inference.
4.2.1.1 Type inference for the edited RO

The first level of type inference involves the RO X whose formula has just been edited. For this first level, three cases must be handled: the RO is a simple cell, the RO is a matrix, and the RO is an abstraction box. Each of these cases is handled by an axiom in Djang et al. [1998]. Using the appropriate axiom, type inference is used to determine X's set of guaranteed operations.

If the RO X is a simple cell, then only axioms [GC] and [Gref] apply. X's new reference determines the appropriate axiom. If X references a constant, then the system infers the set of guaranteed operations from axiom [GC]. This information is stored in the global hash table $PrimitiveTypeOperations. This algorithm, Axiom-GC, is presented in Table 4.1. A simple cell also may reference a non-constant. Two situations now arise. First, the reference may be to another RO on a simple form or a VADT form. Second, the reference may be to an RO on a primitive form such as +’s plusC. In either situation, axiom [Gref] is used to infer the appropriate set of guaranteed operations. The difference is that with respect to the latter, the set of guaranteed operations is predefined in the global hash table $PrimitiveFormsG. The second form of reference also may result in the application of axioms [RecB] and [RecC] if the primitive form’s RO referenced is the RO if:id. These algorithms, Axiom-Gref-RO and Axiom-Gref-PrimForm, are presented in Table 4.1.
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Axiom-GC</strong></td>
<td>Given: a simple cell $X$ that references a constant $C$&lt;br&gt;1. Retrieve the set of operation $G(C)$ for the primitive type by accessing $$PrimitiveTypeOperations$ with the primitive type name as the key.&lt;br&gt;2. Set $G(X) = G(C)$.</td>
</tr>
<tr>
<td><strong>Axiom-Gref-RO</strong></td>
<td>Given: an RO $X$ that references an RO $Y$ which is not on a primitive form&lt;br&gt;1. Retrieve $G(Y)$.&lt;br&gt;2. Set $G(X) = G(Y)$.</td>
</tr>
<tr>
<td><strong>Axiom-Gref-PrimForm</strong></td>
<td>Given: an RO $X$ that references an RO $Y$ on a primitive form&lt;br&gt;1. If $Y$ is if:ifD, then:&lt;br&gt;   - If if:ifB is recursive by IsRecursive?, then:&lt;br&gt;     - perform Axiom-RecB.&lt;br&gt;   - If if:ifC is recursive by IsRecursive?, then:&lt;br&gt;     - perform Axiom-RecC.&lt;br&gt;2. Retrieve $G(Y)$.&lt;br&gt;3. Set $G(X) = G(Y)$.</td>
</tr>
</tbody>
</table>

Table 4.1. Type inference algorithms for edited simple cells. Axiom-Gref-RO and Axiom-Gref-PrimForm also are applicable to matrices.
IsRecursive? | Given: an RO X on form F
---|---
1. If X references a constant C, then return false and stop.
2. If X references an RO Y, then perform the following:
   Let X ← G:Y and let F_i be another form with the same modelName as F for all i = 1…n. If G = F_i for some i and X = Y, then return true and stop. Otherwise, consider G’s defSet entries of format H_i(defSet_i):ref_i at all nesting levels. If F_i:X = ref_i for any i, then return true and stop. Otherwise, return false and stop.
3. If X references an operation Z on a primitive form, perform IsRecursive? on the operands on Z’s primitive form.

Table 4.1 (Continued).

If the RO is a matrix, then axioms [GM] and [Gref] apply. Axiom [GM] is used when the matrix has no formula, or when some of its plain cells have overriding formulas. Axiom [Gref] is used when the matrix’s formula references another matrix. Table 4.2 presents these algorithms (for algorithm Axiom-Gref-RO, the reader is referred to Table 4.1).
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axiom-GM</td>
<td>Given: a matrix RO X that does not reference another matrix or that contains cells with overriding formulas</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve a list of matrix plain cells existing at formula edit time.</td>
</tr>
<tr>
<td></td>
<td>2. For each matrix plain cell, retrieve its set of guaranteed operations and put the set on a list L.</td>
</tr>
<tr>
<td></td>
<td>3. If L is empty, then set G(X) = {} and stop.</td>
</tr>
<tr>
<td></td>
<td>4. If L contains only one set, G(E), then set G(X) = G(E) and stop.</td>
</tr>
<tr>
<td></td>
<td>5. Infer G(X) by performing Intersection with L.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intersection</th>
<th>Given: a list L of 2 or more sets of guaranteed operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Determine the smallest guaranteed set, G(S), in L using a linear time search algorithm.</td>
</tr>
<tr>
<td></td>
<td>2. If G(S) is empty, then return an empty set and stop.</td>
</tr>
<tr>
<td></td>
<td>3. Copy the smallest set and name it returnSet.</td>
</tr>
<tr>
<td></td>
<td>4. For each of the remaining sets (i.e., L – {G(S)}), perform the following steps:</td>
</tr>
<tr>
<td></td>
<td>If returnSet is empty, then step four is completed.</td>
</tr>
<tr>
<td></td>
<td>If this set is G(S), then skip this set.</td>
</tr>
<tr>
<td></td>
<td>For each operation in returnSet, perform the following steps:</td>
</tr>
<tr>
<td></td>
<td>If this operation is in the current set, then continue to the next operation. Otherwise, remove this operation from returnSet and continue to the next operation.</td>
</tr>
<tr>
<td></td>
<td>5. Return returnSet.</td>
</tr>
</tbody>
</table>

Table 4.2. Type inference algorithms for edited matrices. Axiom-Gref-RO is defined in Table 4.1.

If the RO is an abstraction box, then the axiom [GA’] applies. According to axiom [GA’], an abstraction box guarantees all of the operations on its VADT form. Even when an abstraction box is given a formula, it continues to guarantee all the

5 Identical sets of guaranteed operations are shared by ROs and are tested to be the same.
operations on its form. The abstraction box’s guarantee set is updated only when an operation is added to the VADT form, removed from the VADT form or renamed. These situations are examined in Sections 4.2.2 and 4.2.3.

After each respective algorithm is performed, the system determines whether or not the RO is on a VADT form. If the RO is on a VADT form, the system checks if the form’s defSet has been changed. For this situation, the system must add the operation to any affected sets of required operations, using axiom [R1]. This algorithm is described in Section 4.2.2 (see algorithm Axiom-R1bc-add).

4.2.1.2 Type inference for affecting ROs

When an RO X’s formula is changed, two type inference situations arise for ROs that affect X. For all ROs Y that directly or indirectly affected X before X’s formula edit, their requirements must be un-united from R(X). For all ROs Z that now directly or indirectly affect X after X’s formula edit, their requirements must be united with R(X). In either case, no guarantees need to be inferred because no formulas have been edited. In our type system, the un-unioning for Y is performed before the unioning for Z. For the purposes of clarity and understanding, however, we first describe the union processing for Z.

For inferring the requirements of Z, the type system uses axioms [R1], [RM] and [RN]. Our implementation divides axiom [R1] into its three components referred to as axioms [R1a], [R1b] and [R1c]. Axiom [R1a] states that the requirements of RO Z include the requirements of ROs referencing Z. That is, for X₁, X₂, ... Xₙ ← Z, R(Z) = ∪ i=1..n of R(Xᵢ). Axiom [R1b] states that the requirements of RO Z include the operations being referenced on all copies of Z’s VADT form. In axiom [R1c], the requirements of RO Z include any additional operations on all copies of Z’s

---

6 If an abstraction box is given a reference to an RO that is a different type, an impersonation error occurs. This error is caught by the generalization engine in Forms/3. For the type system, handling this error is unnecessary and is redundant.
VADT form that are in the defSets defining the form copy. For our purposes in this section, only axiom [R1a] is applicable because axioms [R1b] and [R1c] involve the addition and removal of operations from sets of required operations. These two axioms are discussed in Section 4.2.2. Table 4.3 presents the algorithms involved in inferring R(Z).

The algorithms in Table 4.3 are the first examples of "methodized" algorithms, i.e., in our object-oriented implementation, the algorithms are implemented as different methods for Forms/3’s different kinds of ROs. In propagating the changes to the different sets of required operations, the type system has to account for the different ROs such as matrix plain cells and simple cells. Matrix plain cells often are handled differently than the other ROs because they affect the guarantees and requirements of their parent matrix. Therefore, when a matrix plain cell’s guarantees or requirements change, the type system must propagate this change to the cell’s parent matrix. Henceforth, since most of the methods for one algorithm are similar, only those methods which are significantly different from the others will be presented in future tables with the remaining methods presented in Appendix A. Algorithms that describe propagating set changes recursively also are located in Appendix A. These algorithms do not provide any additional insight into our type system and often are identified by the suffix "-Rec" or the prefix "Propagate-".
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
</table>
| Propagate-Requirements | Given: an RO X and an abstraction box Z that X references  
1. Retrieve R(X).  
2. Perform Axiom-R1a on Z and R(X).  
3. If Z is on a VADT form, then perform Axiom-R1bc-add.  
4. Retrieve the RO W referenced by Z.  
5. Perform Propagate-Requirements-Rec on Z and W.  

Given: an RO X and a matrix size cell Z that X references  
1. If Z is on a VADT form, then perform Axiom-R1bc-add.  
2. Retrieve the RO W referenced by Z.  
3. Perform Propagate-Requirements-Rec on Z and W.  

Given: an RO X and a matrix plain cell Z that X references  
1. Retrieve R(X).  
2. Retrieve Z's parent matrix M.  
3. Retrieve R(M).  
5. Perform Union-Decr on R(M) and R(Z).  
6. Perform Axiom-R1a on Z and R(X).  
7. If Z is on a VADT form, then perform Axiom-R1bc-add.  
9. Perform Union-Incr on R(M) and R(Z).  
10. Retrieve the RO W referenced by Z.  
11. Perform Propagate-Requirements-Rec on Z and W.  

Given: an RO X and a simple cell Z that X references  
1. Retrieve R(X).  
2. Perform Axiom-R1a on Z and R(X).  
3. If Z is on a VADT form, then perform Axiom-R1bc-add.  
4. Retrieve the RO W referenced by Z.  
5. Perform Propagate-Requirements-Rec on Z and W.  

Axiom-RM  
| Given: a matrix X  
| 1. Retrieve a list of matrix plain cells for X.  
2. Retrieve each cell's requirements and place the set on the list L, e.g. given R(X) = {push, pop}, L = {{push, pop}}  
3. If L is empty, set R(X) = {} and stop.  
4. If L contains only one set R(E), then set R(X) = R(E) and stop.  
5. Infer R(X) by performing Union on L.  

<p>| Table 4.3. Type inference algorithms for currently affecting ROs Z. |</p>
<table>
<thead>
<tr>
<th>Axiom-R1a</th>
<th>Given: an RO Z and R(X) such that X references Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Retrieve R(Z).</td>
</tr>
<tr>
<td></td>
<td>2. Perform Union-Incr on R(Z) and R(X).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union-Incr</th>
<th>Given: R(Z) and R(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. For each op ∈ R(X), perform the following step:</td>
</tr>
<tr>
<td></td>
<td>If op ∈ R(Z), then increment op’s counter.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, add op with counter = 1 to R(Z).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union⁷</th>
<th>Given: a list L of two or more sets of required operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Determine the largest set R(B) in L using a linear time search algorithm.</td>
</tr>
<tr>
<td></td>
<td>2. If R(B) is empty, then return an empty set and stop.</td>
</tr>
<tr>
<td></td>
<td>3. Make a copy, returnSet, of R(B) with each operation’s counter set to 1.</td>
</tr>
<tr>
<td></td>
<td>4. For each set on the list excluding R(B) (i.e., L = {R(B)}), perform the following steps:</td>
</tr>
<tr>
<td></td>
<td>For each operation, op, in the current set, perform the following step:</td>
</tr>
<tr>
<td></td>
<td>If op ∈ returnSet, then increment the counter in returnSet. Otherwise, add the operation to returnSet with the counter set to 1.</td>
</tr>
<tr>
<td></td>
<td>5. Return returnSet.</td>
</tr>
</tbody>
</table>

Table 4.3 (Continued).

The type inference algorithms for the requirements of the ROs Y that must be un-unioned from R(X) are based on axioms [RI], [RM] and [RN]. Table 4.4 presents these algorithms.

⁷The set union find algorithm described by Aho, Hopcroft and Ullman [1974] is not used in our research because their algorithm involves the union of disjoint sets. The sets of required operations in our type system are not disjoint sets.
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Undo-Ref</strong></td>
<td>Given: a simple cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve R(Y).</td>
</tr>
<tr>
<td></td>
<td>2. If R(Y) = {}, then stop.</td>
</tr>
<tr>
<td></td>
<td>3. If Y is referenced by another RO W (i.e., Y ← W), then perform Undo-Ref-Propagate on W and R(U).</td>
</tr>
<tr>
<td></td>
<td>4. Perform Union-Decr on R(Y) and R(U).</td>
</tr>
<tr>
<td></td>
<td>5. If Y is on a VADT form, then perform Undo-Ref-VADT.</td>
</tr>
<tr>
<td><strong>Undo-Ref-VADT</strong></td>
<td>Given: an RO Y and R(U) where R(U) is a requirements set to be un-unioned</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve Y's VADT form FY.</td>
</tr>
<tr>
<td></td>
<td>2. If Y is in FY's defSet, then stop.</td>
</tr>
<tr>
<td></td>
<td>3. Retrieve a list L of ROs that reference Y.</td>
</tr>
<tr>
<td></td>
<td>4. If L is not empty, then stop.</td>
</tr>
<tr>
<td></td>
<td>5. Perform Axiom-R1bc-remove on Y (see Table 4.8).</td>
</tr>
<tr>
<td><strong>Union-Decr</strong></td>
<td>Given: R(Y) and R(U) where R(U) contains the set of operations to be un-unioned from R(Y)</td>
</tr>
<tr>
<td></td>
<td>1. For each operation in R(Y), perform the following steps:</td>
</tr>
<tr>
<td></td>
<td>Find the operation in R(Y).</td>
</tr>
<tr>
<td></td>
<td>Retrieve the operation counter in R(Y).</td>
</tr>
<tr>
<td></td>
<td>Decrement the counter by 1.</td>
</tr>
<tr>
<td></td>
<td>If the counter equals zero, then remove the operation from R(Y). Otherwise, place the new counter in R(Y).</td>
</tr>
</tbody>
</table>

Table 4.4. Type inference algorithms for formerly affecting ROs Y.

4.2.1.3 Type inference for affected ROs

When an RO X's formula is edited, all ROs V affected by X (i.e., all ROs V that reference X directly or indirectly) are affected with respect to their sets of guaranteed operations. Two cases arise. First, if the affected RO V is a matrix plain cell, then the parent matrix's set of guaranteed operations also must be inferred. Otherwise,
V's set of guaranteed operations can be set equal to X's set of guaranteed operations. The algorithm, Propagate-Guarantees, is presented in Table 4.5.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate-Guarantees</td>
<td>Given: an RO X whose formula has just been edited</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve the list of ROs L that reference X.</td>
</tr>
<tr>
<td></td>
<td>2. For each RO V in L, perform Propagate-Guarantees-Rec on V and X.</td>
</tr>
<tr>
<td>Propagate-Guarantees-Rec</td>
<td>Given: a simple cell V and an RO X that V references</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve G(V).</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve G(X).</td>
</tr>
<tr>
<td></td>
<td>3. Set G(V) = G(X).</td>
</tr>
<tr>
<td></td>
<td>4. Perform Propagate-Guarantees on V.</td>
</tr>
</tbody>
</table>

Table 4.5. Type inference algorithm for affected ROs.

4.2.2 Adding/Removing Operations

The only manner other than through propagation in which an operation can be added to a set of guaranteed operations is to add an operation to a VADT form. Similarly, an operation can only be removed by removing it from its VADT form. For guaranteed sets, when an operation is created on a VADT form, it is added to the main abstraction box's set of guaranteed operations. When an operation is removed, it also is removed from the abstraction box's set of guaranteed operations. Afterwards, in both cases, the new set information is propagated to all affected ROs. For most of these ROs, however, their sets of guaranteed operations are not altered.

Propagating guarantees to all ROs V is necessary—even though in many cases they share the same hash table—in order to ensure that the sets of guaranteed operations for matrices are updated when their plain cells are updated.
because they share the same hash table as the main abstraction box. The only ROs that are affected are those ROs on primitive forms and matrices. For these ROs, the appropriate type inference algorithm is used to infer their new sets of guaranteed operations. Table 4.6 contains the algorithms for adding and removing a guaranteed operation.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axiom-GA'-add</td>
<td><strong>Given: an RO X that is added to a VADT form F_Q</strong></td>
</tr>
<tr>
<td></td>
<td>1. Retrieve X's VADT form F_Q.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve F_Q's main abstraction box ABS.</td>
</tr>
<tr>
<td></td>
<td>4. Add X's name to G(ABS).</td>
</tr>
<tr>
<td></td>
<td>5. Retrieve a list L of abstraction boxes on F_Q.</td>
</tr>
<tr>
<td></td>
<td>6. For each abstraction box ABS_i on L, perform</td>
</tr>
<tr>
<td></td>
<td>Propagate-Add-Guarantee-Operation on ABS_i and X.</td>
</tr>
<tr>
<td>Axiom-GA'-remove</td>
<td><strong>Given: an RO X that is removed from a VADT form F_Q</strong></td>
</tr>
<tr>
<td></td>
<td>1. Retrieve X's VADT form F_Q.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve F_Q's main abstraction box ABS.</td>
</tr>
<tr>
<td></td>
<td>4. Remove X from G(ABS).</td>
</tr>
<tr>
<td></td>
<td>5. Retrieve a list L of abstraction boxes on F_Q.</td>
</tr>
<tr>
<td></td>
<td>6. For each abstraction box ABS_i on L, perform</td>
</tr>
<tr>
<td></td>
<td>Propagate-Remove-Guarantees on ABS_i and X.</td>
</tr>
</tbody>
</table>

Table 4.6. Algorithms for adding and removing guaranteed operations.

In order for an operation to be added to a set of required operations, an RO V that is not on the operation's VADT form has to reference the operation, or the

---

9 The type system must propagate through these ROs because primitive form operations and matrices sometimes do not share hash tables with other ROs. In order to ensure that primitive form operations and matrices are updated, the type system must propagate the changes to all affected ROs.
operation has to be a member of a VADT form’s defSet. The operation is removed when neither condition exists. The algorithms are presented in Table 4.7.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axiom-R1bc-add</td>
<td>Given: an RO X that is on a VADT form FQ and either is referenced by an RO V not on FQ or is a member of FQ’s defSet</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve X’s VADT form FQ.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve the main abstraction box ABS on FQ.</td>
</tr>
<tr>
<td></td>
<td>3. Retrieve the RO Y referenced by ABS.</td>
</tr>
<tr>
<td></td>
<td>4. Retrieve R(Y).</td>
</tr>
<tr>
<td></td>
<td>5. If X ∈ R(Y), then stop.</td>
</tr>
<tr>
<td></td>
<td>6. Add X to R(Y).</td>
</tr>
<tr>
<td></td>
<td>7. Perform Propagate-Requirements-Rec.</td>
</tr>
<tr>
<td>Axiom-R1bc-remove</td>
<td>Given: an RO X that is on a VADT form FQ, is not referenced by an RO V not on FQ, and is not a member of FQ’s defSet</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve X’s VADT form FQ.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve the main abstraction box ABS on FQ.</td>
</tr>
<tr>
<td></td>
<td>3. Retrieve the RO Y referenced by ABS.</td>
</tr>
<tr>
<td></td>
<td>4. Retrieve R(Y).</td>
</tr>
<tr>
<td></td>
<td>5. Remove X from R(Y).</td>
</tr>
</tbody>
</table>

Table 4.7. Algorithms for adding and removing required operations.

4.2.3 Renaming Operations

As mentioned in the beginning, each operation on a VADT form except for the main abstraction box and the Image cell starts out with its cellID as its default operation name. In this section, we discuss the algorithms that handle renaming an operation. Renaming simple form ROs is not handled because these ROs are not introduced as operations into any sets of guaranteed and required operations. Therefore, name changes for these ROs do not affect the type system.
When an operation on a VADT form is renamed, the sets of guaranteed and required operations are affected. With respect to sets of guaranteed operations, four cases arise. In the first case, the operation is on a copied VADT form. In the second case, the operation is a copy of an operation on another VADT form. For these two cases, the type system replaces the old name with the new name and adds the name “like <old operation name>” to the guaranteed sets of all affected ROs. In the third case, the operation is not a copied operation. In the fourth case, the operation originally was a copied operation, but the user has edited its formula, in order to override the formula dependency between it and the original operation. For these two cases, the system only has to replace the old name with the new name in all affected guaranteed sets.

Sets of required operations are affected by an operation name change if that operation is involved in a polymorphic type reference, or if the operation is in its VADT form’s defSet. Otherwise, the operation is not a member of any RO’s set of required operations, and its name change will not affect any RO’s required set of operations. Table 4.8 presents the algorithms used when an operation on a VADT form is renamed.
**Rename-Guarantees**

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename-Guarantees</td>
<td>Given: an RO X that is renamed and is on a VADT form ( F_X ), and its old name</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve ( X )'s VADT form ( F_X ).</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve the main abstraction box ( \text{ABS} ) on ( F_X ).</td>
</tr>
<tr>
<td></td>
<td>3. Retrieve ( G(\text{ABS}) ).</td>
</tr>
<tr>
<td></td>
<td>4. Remove the old operation name from ( G(\text{ABS}) ).</td>
</tr>
<tr>
<td></td>
<td>5. Add ( X )'s name to ( G(\text{ABS}) ).</td>
</tr>
<tr>
<td></td>
<td>6. If ( X ) is a copied RO whose model RO is on another VADT form, then add the operation name “like &lt;old name&gt;” to ( G(\text{ABS}) ).</td>
</tr>
<tr>
<td></td>
<td>7. Retrieve a list ( L ) of abstraction boxes on ( F_X ).</td>
</tr>
<tr>
<td></td>
<td>8. For each abstraction box ( \text{ABS}_j ) on ( L ), perform Propagate-NameChange-Guarantees on ( \text{ABS}_i ), ( X ), old name and like &lt;old name&gt;.</td>
</tr>
</tbody>
</table>

**Rename-Requirements**

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename-Requirements</td>
<td>Given: an RO X that is renamed and is on a VADT form ( F_X ) and its old name</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve ( X )'s VADT form ( F_X ).</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve the main abstraction box ( \text{ABS} ) on ( F_X ).</td>
</tr>
<tr>
<td></td>
<td>3. If ( X ) is not referenced by an RO not on ( F_X ), then:</td>
</tr>
<tr>
<td></td>
<td>a. If ( X ) is not in ( F_X )'s defSet, then stop.</td>
</tr>
<tr>
<td></td>
<td>Otherwise,</td>
</tr>
<tr>
<td></td>
<td>a. Retrieve the RO ( Y ) referenced by ( \text{ABS} ).</td>
</tr>
<tr>
<td></td>
<td>b. Retrieve ( R(Y) ).</td>
</tr>
<tr>
<td></td>
<td>c. Retrieve the counter for old name in ( R(Y) ).</td>
</tr>
<tr>
<td></td>
<td>d. Remove old name from ( R(Y) ).</td>
</tr>
<tr>
<td></td>
<td>e. Add ( X )'s name to ( R(Y) ) with the old counter.</td>
</tr>
<tr>
<td></td>
<td>f. Perform Propagate-NameChange-Requirements on ( Y ), ( X ) and old name.</td>
</tr>
</tbody>
</table>

Table 4.8. Algorithms for renaming VADT form ROs.

### 4.2.4 Type Checking

After an RO's formula is edited, type checking is performed on the following kinds of ROs: the RO \( X \) whose formula has been edited, those ROs \( Z \) that are now referenced by \( X \), and those ROs \( V \) that directly or indirectly reference \( X \). Type
checking is not performed on those ROs Y that were formerly referenced by X because Y’s un-unioned requirements set (see algorithm Undo-Ref) is a subset of its previous set, and therefore, R(Y) is still a subset of G(Y).

Type checking is performed after an RO’s set of guaranteed or required operations has been inferred. The type checking algorithm is derived from the model’s definition of a type safe program. An RO X is type safe if and only if R(X) ⊆ G(X). Algorithm Type-Check is presented in Table 4.9.

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-Check</td>
<td>Given: an RO X and a boolean variable typeSafe?</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve R(X).</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve G(X).</td>
</tr>
<tr>
<td></td>
<td>3. If R(X) = {}, then return typeSafe? = T and stop.</td>
</tr>
<tr>
<td></td>
<td>4. If</td>
</tr>
<tr>
<td></td>
<td>5. Perform the following step for each operation in R(X):</td>
</tr>
<tr>
<td></td>
<td>If the operation is not in G(X), then return typeSafe? = F and stop.</td>
</tr>
<tr>
<td></td>
<td>4. Return typeSafe? = T.</td>
</tr>
</tbody>
</table>

Table 4.9. Type check algorithm.

4.3 Examples of Type Inference and Type Checking

In this section, we present three examples of the type inference and type checking algorithms. The examples are based on the Stack example presented in Chapter 3 (see Figure 3.1). In these examples, the programmer adds a new simple cell to the form Stack that references the bottom of the stack and names it “bottom”. The three examples include: adding an RO to a VADT form, providing an RO with a new formula and renaming a pre-existing VADT form RO.
4.3.1 Example: Type Inference and Type Checking After an RO is Added to a VADT Form

Using the Stack defined in Figure 3.1, the programmer now adds a simple cell to the form. The cell's cellID is "cell-123", and its name defaults to "cell-123". Axiom-GA'-add is used by the system to add the "cell-123" to all affected sets of guaranteed operations. First, the form Stack is retrieved. Then the abstraction box Stack is retrieved as well as its guarantees, G(Stack:Stack). "cell-123" is added to G(Stack:Stack). Since no type error existed prior to the addition, no type checking needs to be performed because the addition of a new operation to the set of guaranteed operations cannot invalidate the previous relationship, R(Stack:Stack) ⊆ G(Stack:Stack). If a type error did exist before the addition, the system would perform Type-Check on Stack:Stack, in order to determine whether or not a type error still existed.

Afterwards, each abstraction box on the form Stack is placed in a list L such that L = {Stack, push, pop}. Using the contents of L, Propagate-Guarantees then propagates the new guarantees information to all affected ROs. No ROs, however, reference any of these three abstraction boxes, and no propagation is performed in this example. The type system does not propagate guarantees information to push and pop because these abstraction boxes share their sets of guaranteed operations with Stack.

4.3.2 Example: Type Inference and Type Checking After a Simple Cell Formula Edit

The programmer now decides to give cell-123 the formula "Stack[items][1@1]" which is a reference to the first element on the stack. As mentioned in Section 4.2.1, three levels of type inference occur. The first step is to undo the requirements of all
the ROs Y that used to directly or indirectly affect cell-123. The system does not need to perform this first step because cell-123 had no previous RO reference.

The next step is to infer cell-123’s new set of guaranteed operations and to type check cell-123. Because cell-123 references another RO that is not on a primitive form, the type system uses Axiom-Gref-RO on cell-123 and Stack[items][1@1].

G(Stack[items][1@1]) is retrieved, and the system sets G(cell-123) = G(Stack[items][1@1]). Since G(Stack[items][1@1]) = primitiveTextOperations, G(cell-123) points to the same hash table and the same set of primitiveTextOperations. The system then performs Type-Check on cell-123. The system retrieves R(cell-123) and G(cell-123). Since R(cell-123) = {} and hence R(cell-123) ⊆ G(cell-123), cell-123 is type safe.

The final step is to propagate the new formula to all affected ROs. This final step is divided into propagating requirements and propagating guarantees. In order to propagate requirements, the system performs Propagate-Requirements with cell-123 and Stack[items][1@1]. The type system uses the methodized algorithm Propagate-Requirements for matrix plain cells because Stack[items][1@1] is a matrix plain cell. The system retrieves R(cell-123). Since R(cell-123) = {} and no sets of required operations will be affected, no further propagation is necessary. In order to propagate guarantees, the system uses algorithm Propagate-Guarantees with cell-123. The list of ROs L is empty and no direct propagation is required. The program has now been type checked and is type safe.

4.3.3 Example: Type Inference and Type Checking After a VADT Form’s RO is Renamed

Finally, the programmer renames cell-123 to “bottom”. When an RO on a VADT form is renamed, both algorithms Rename-Guarantees and Rename-Requirements are performed on cell-123 and its old name “cell-123”, in order to ensure that all affected sets are updated. In Rename-Guarantees, the form Stack is retrieved. Then the abstraction box Stack is retrieved as well as its set of guaranteed
operations. The type system then removes “cell-123” from \(G(\text{Stack:Stack})\) and adds “bottom” to \(G(\text{Stack:Stack})\). Since cell-123 is not a copied RO, no “like” operation is added to \(G(\text{Stack:Stack})\). The type system retrieves a list \(L\) that contains the abstraction boxes \(\text{Stack, push and pop}\). No ROs reference these abstraction boxes, so no further propagation is necessary.

In Rename-Requirements, the form \(\text{Stack}\) and the abstraction box \(\text{Stack}\) are retrieved. Since no RO that is not on the form \(\text{Stack}\) references cell-123 and cell-123 is not in the form \(\text{Stack}\)’s defSet, no propagation to sets of required operations is necessary. At this point, the program is type checked and is type safe.

### 4.4 Space and Time Analyses

In this section, the space and time analyses for our data structures and algorithms are presented. Space complexities for the data structures are first introduced. Afterwards, time complexities for the algorithms are discussed.

#### 4.4.1 Space Analyses

The space analyses in this section are presented as follows: first, the space complexity of a single RO is derived; and from this derivation, we obtain the space complexities for a Forms/3 program. The notational conventions used in this section are given in Table 4.10.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>numSimpleForms</td>
<td>number of simple forms</td>
</tr>
<tr>
<td>numSimpleFormROs</td>
<td>number of ROs on a simple form</td>
</tr>
<tr>
<td>numVADTs</td>
<td>number of VADT forms</td>
</tr>
<tr>
<td>numVADTROs</td>
<td>number of operations on a VADT form</td>
</tr>
<tr>
<td>numMatrixCells</td>
<td>number of matrix plain cells</td>
</tr>
<tr>
<td>numForms</td>
<td>numSimpleForms + numVADTs, i.e., total number of forms</td>
</tr>
<tr>
<td>numROs</td>
<td>numSimpleForms* numSimpleFormROs + numVADTs * numVADTROs, i.e., total number of ROs</td>
</tr>
<tr>
<td>P</td>
<td>a Forms/3 program</td>
</tr>
<tr>
<td>X</td>
<td>a given RO</td>
</tr>
</tbody>
</table>

Table 4.10. Notational conventions for space and time analyses.

A single RO is composed of a set of guaranteed operations and a set of required operations. In the worst case, the space required for a set of guaranteed operations is O(numVADTs * numVADTROs). The worst case is a contrived example because it only occurs when each VADT form is a copy of the previous VADT form, and with all of each copied form's operations renamed (but not edited). In this example, no sets of guaranteed operations are shared because each VADT form is a new type definition with renamed operations. If each operation on each VADT form, however, was edited, then the space requirements fall to O(numVADTROs) because the long list of "like-a" guaranteed operations would disappear. The worst case situation is unlikely to occur because new type definitions often involve some new operation formulas. For example, in the Stack and Queue examples presented in Chapter 3, Queue is copied from Stack. The programmer, however, edits Queue:enqueue's formula, thereby eliminating the "like-a" relationship between Queue:enqueue and Stack:push.

For a set of required operations, the worst case occurs when an RO is required to support all operations in a program. In other words, the space required for a set of required operations is O(numVADTs * numVADTROs). This situation may arise
for a matrix when each cell in the matrix is required to support a different set of operations. The worst case situation is unlikely to occur because in order for an RO to be required to support all the operations on its VADT form, either each operation on its VADT form must be involved in a polymorphic reference or must be in the form's defSet. In the Stack collection example presented in Figure 3.3, collection is required to support only the operations top and push. It is not required to support all of Stack's operations. Many operations on a VADT form are "private" operations that may never be involved in a polymorphic reference. For example, on the form Stack, the operations Image and lines can be considered private operations that only describe a Stack's visual appearance.

For a single RO, the space required for both sets of operations is \(O(\text{numVADTs} \times \text{numVADTROs})\). A Forms/3 program \(P\) has \(\text{numSimpleForms}\) simple forms and \(\text{numVADTs}\) VADT forms. If each simple form has \(\text{numSimpleFormROs}\) ROs and each VADT form has \(\text{numVADTROs}\) ROs, then \(P\) has \(O(\text{numSimpleForms} \times \text{numSimpleFormROs} + \text{numVADTs} \times \text{numVADTROs})\) ROs. That is, in the worst case, \(P\) is equivalent to the number of ROs in the program multiplied by each RO's space. The amount of space required for all sets of guaranteed operations is \(O(\text{numVADTs}^2 \times \text{numVADTROs}^2 + \text{numSimpleForms} \times \text{numSimpleFormROs} \times \text{numVADTs} \times \text{numVADTROs})\). The amount of space required for all sets of required operations also is \(O(\text{numVADTs}^2 \times \text{numVADTROs}^2 + \text{numSimpleForms} \times \text{numSimpleFormROs} \times \text{numVADTs} \times \text{numVADTROs})\). Therefore, for \(P\), the space required for both sets of operations also is \(O(\text{numVADTs}^2 \times \text{numVADTROs}^2 + \text{numSimpleForms} \times \text{numSimpleFormROs} \times \text{numVADTs} \times \text{numVADTROs})\). Table 4.11 summarizes the space requirements presented in this section.

The worst case situation is unlikely to occur. It only occurs when a programmer creates \(\text{numVADTs}\) types (with \(\text{numVADTROs}\) abstraction boxes on each form) that are identical except for each type its respective operations have been renamed. The programmer then creates \(\text{numMatrixCells}\), and each matrix plain cell references a different type. The programmer also creates \(\text{numSimpleFormROs} - \text{numMatrixCells}\)
ROs, each of which reference a different operation on VADT form. The result is that each matrix plain cell is required to support all the operations for its type.

In our type system, whenever an RO X references an RO Z, both X and Z share the same set of guaranteed operations, thereby conserving space. For example, in the Stack operation example presented in Figure 3.3, the cells collection and the-rest share the same hash table as Stack:Stack, Stack:push, Stack:pop, 348-Stack:Stack, 348-Stack:push and 348-Stack:pop. The cell removed-item shares the same hash table as Stack:top, 348-Stack:top, Stack[items][1@lastcol] and 348-Stack[items][1@lastcol]. For all the operations on the VADT form 348-Stack, each operation shares its set of guaranteed operations with its respective model operation on the VADT form Stack.

Regarding sets of required operations, the population study example illustrates our previous point that matrices are unlikely to support all the operations in a program. The matrix location is required to support only primitiveTextOperations, the matrix population is required to support only primitiveNumberOperations, and the matrix graph is required to support only primitiveCircleOperations.

<table>
<thead>
<tr>
<th>Set Description</th>
<th>Worst Case Space Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of guaranteed operations for an RO</td>
<td>O(numVADTs * numVADTROs)</td>
</tr>
<tr>
<td>Set of required operations for an RO</td>
<td>O(numVADTs * numVADTROs)</td>
</tr>
<tr>
<td>Set of guaranteed operations for P</td>
<td>O(numVADTs^2 * numSimpleFormROs^2 + numSimpleForms * numSimpleFormROs * numVADTs * numVADTROs)</td>
</tr>
<tr>
<td>Set of required operations for P</td>
<td>O(numVADTs^2 * numSimpleFormROs^2 + numSimpleForms * numSimpleFormROs * numVADTs * numVADTROs)</td>
</tr>
</tbody>
</table>

Table 4.11. Summary of space analyses.
4.4.2 Time Analyses

The time analyses are presented in the order that the algorithms were presented. In estimating these time analyses, we assumed that searching a hash table is $O(1)$ (see discussion in Section 4.1.2). The notational conventions used in this section are the same as in Table 4.10.

4.4.2.1 Type inference

When an RO $X$'s formula is edited, the type system performs four main steps. First, all previously referenced ROs $Y$ (directly and indirectly) have their requirements un-unioned from $R(X)$. Second, $G(X)$ is inferred. Third, all directly and indirectly referenced ROs $Z$ have their requirements unioned with $R(X)$. Fourth, $G(X)$ is propagated to all affected ROs $V$. The time complexity for each of these steps depends on the ROs involved. Most of the time, if matrix plain cells are involved in the type inference, the time complexity is multiplied by a magnitude of $O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$ because each matrix plain cell's matrix must have its guarantees inferred. For example, in the fourth step, if matrix plain cells are involved in the propagation, the time complexity is $O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$, where $\text{numMatrixCells} = \text{numROs} - 4$. If matrix plain cells are not involved, the time complexity is $O(\text{numROs})$.

The time complexity for the first step is $O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$ because Undo-Ref is the algorithm used. This situation arises when a chain of references exists. The type system propagates through each affecting RO (i.e., $Y_i \leftarrow X$, where $i = 1 \ldots \text{numROs}$) and performs Union-Decr on each RO's requirements. Union-Decr has the time complexity of $O(\text{numVADTs} \times \text{numVADTROs})$ because it is the un-union of only two sets of operations. In the worst case, a set of requirements has $O(\text{numVADTs} \times \text{numVADTROs})$ operations.
For the second step, most of these algorithms are $O(1)$ because only pointers are set to hash tables, and no searching is involved in the algorithms. For example, when the type system uses Axiom-GC, the system retrieves $G(C)$ and sets the pointer in $X$'s :guaranteedOps slot to the hash table representing $G(C)$. Axiom-Gref-PrimForm has a time complexity of $O(\text{numVADTs} \times \text{numVADTROs})$ because at worst, Intersection is performed on only two sets of operations. If the referenced RO is an instance of if:iD, i.e., the referenced RO is the result RO ifD on the primitive form if, the time complexity still is $O(\text{numVADTs} \times \text{numVADTROs})$ because this time complexity outweighs the complexity arising from IsRecursive?.

IsRecursive?'s time complexity of $O(\max(\text{numSimpleFormROs}, \text{numVADTROs}))$ arises from the fact that in the worst case the system recursively searches the ROs on a form for recursion.

The third step has a time complexity of $O(n\text{umROs} \times \text{numVADTs} \times \text{numVADTROs})$ because Propagate-Requirements is the algorithm initially called. The worst case scenario for this step is similar to the situation in the first step. The only difference is that Axiom-R1a is called, instead of Union-Decr. Axiom-R1a then calls Union-Incr to union only two sets together. The union of two sets is $O(\text{numVADTs} \times \text{numVADTROs})$ because at worse a matrix may be required to support all the operations in the program.

In the final step, Propagate-Guarantees is called, and the resulting time complexity is $O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$ where numMatrixCells = numROs – 4. The worst case situation arises when RO X is referenced by numMatrixCells matrix plain cells, all of which reside in the same matrix. For each matrix plain cell, Axiom-GM is performed on its parent matrix. Axiom-GM in turn calls Intersection. Intersection performs in $O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$ because the intersection of all the sets of guaranteed operations for all the matrix plain cells in a matrix must be derived. Therefore, since Axiom-GM is performed for each affected matrix plain cell, Propagate-Guarantees has the time complexity of $O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$. 
Thus, when an RO's formula is edited, the total time complexity of type inference is $O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$, where $\text{numMatrixCells} = \text{numROs} - 4$. The worst case situation is dominated by the fourth step's worst case scenario. This worst case situation occurs when matrix plain cells are involved. If matrix plain cells are not involved, the total time complexity of type inference is $O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$ because sets of required operations need to be inferred. The time complexities for the type inference algorithms are presented in Table 4.12.

<table>
<thead>
<tr>
<th>User Action</th>
<th>Algorithm</th>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>formula edit</td>
<td>Axiom-GC</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-Gref-RO</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-Gref-PrimForm</td>
<td>$O(\text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-RecB</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-RecC</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>formula edit</td>
<td>IsRecursive?</td>
<td>$O(\max(\text{numSimpleROs}, \text{numVADTROs}))$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-GM</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Intersection</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Propagate-Requirements</td>
<td>$O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-RM</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Axiom-R1a</td>
<td>$O(\text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Union-Incr</td>
<td>$O(\text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Union</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Undo-Ref</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$ where $\text{numMatrixCells} = \text{numROs} - 3$</td>
</tr>
<tr>
<td>formula edit</td>
<td>Undo-Ref-VADT</td>
<td>$O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
</tbody>
</table>

Table 4.12. Worst case time complexities for type inference algorithms.
Table 4.12 (Continued).

4.4.2.2 Adding/removing operations

The worst case situation for adding and removing operations often involves propagating set changes to matrices and primitive form operations such as append because the algorithms for these propagations include Intersection or Union. For adding and removing a guaranteed operation, the best case occurs when a VADT form contains only the main abstraction box and no ROs reference the abstraction box. In this case, the type inference algorithms are \( O(1) \). The worst case situation occurs when each of the matrix plain cells in a matrix reference the same abstraction box, and a new RO is added or removed from the abstraction box’s form. In this situation, adding a new operation is \( O(\text{numMatrixCells}^2) \) because for each matrix plain cell, its parent matrix is retrieved, the matrix’s list of plain cells is retrieved, and the system checks if the new operation is a member of each cell’s guarantees. The removal of an operation is \( O(\text{numMatrixCells}) \) because the system can immediately remove the operation from the matrix’s set of guaranteed operations.

For adding and removing a required operation, the best case occurs when the new or removed RO is not a member of its VADT form’s defSet and is not referenced by an RO on another form. The algorithms in this case are \( O(1) \). In the worst case, propagation to matrices occurs, and the type inference algorithms are \( O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs}) \). The worst case situation arises when matrix plain cells in separate matrices reference one another. Since each matrix plain cell resides in its own matrix, the algorithms Union-Decr, Axiom-R1a and Union-Incr (all of which are \( O(\text{numVADTs} \times \text{numVADTROs}) \) are performed.
numMatrixCells times, where \( \text{numMatrixCells} = \text{numROs} / 4 - 1 \). These time complexities are summarized in Table 4.13.

<table>
<thead>
<tr>
<th>User Action</th>
<th>Algorithm</th>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add RO to VADT form</td>
<td>Axiom-GA'-add</td>
<td>( O(\text{numMatrixCells}^2) ) where ( \text{numMatrixCells} = \text{numROs} - 4 )</td>
</tr>
<tr>
<td>Remove RO from VADT form</td>
<td>Axiom-GA'-remove</td>
<td>( O(\text{numMatrixCells}) ) where ( \text{numMatrixCells} = \text{numROs} - 4 )</td>
</tr>
<tr>
<td>Add a polymorphic reference or add to a form’s defSet</td>
<td>Axiom-R1bc-add</td>
<td>( O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs}) )</td>
</tr>
<tr>
<td>Remove polymorphic reference</td>
<td>Axiom-R1bc-remove</td>
<td>( O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs}) )</td>
</tr>
</tbody>
</table>

Table 4.13. Worst case time complexities of algorithms for adding and removing operations.

4.4.2.3 Renaming operations

The worst case scenarios for renaming operations is similar to those situations for adding and removing operations—algorithms for propagating to matrices and primitive form operations that require type inference result in larger time complexities. The best case scenarios also are similar and often occur when the type system does not need to propagate any set information.

With respect to guarantees, the worst case situation occurs when each matrix plain cell in a matrix references the same abstraction box, and the abstraction box or the Image cell’s name is changed. Rename-Guarantees has a time complexity of \( O(\text{numMatrixCells}) \) because for each of the numMatrixCells matrix plain cells, the name change is propagated to its guarantees as well as its parent matrix’s guarantees. With respect to requirements, the worst case situation arises when the abstraction box references an RO which in turn references another RO, and so on. Since
requirements are not shared, propagation reaches all affecting ROs, and the time complexity is $O(\text{numROs})$.

Thus, when an operation is renamed, the total time complexity of propagation is $O(\text{numROs})$. The worst case time complexities are presented in Table 4.14.

<table>
<thead>
<tr>
<th>User Action</th>
<th>Algorithm</th>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename RO on VADT form</td>
<td>Rename-Guarantees</td>
<td>$O(\text{numMatrixCells})$ where numMatrixCells = numROs – 4</td>
</tr>
<tr>
<td>Rename RO on VADT form</td>
<td>Rename-Requirements</td>
<td>$O(\text{numROs})$</td>
</tr>
</tbody>
</table>

Table 4.14. Worst case time complexities for renaming operation algorithms.

4.4.2.4 Type checking

Type-Check performs in $O(\text{numVADTs} \times \text{numVADTROs})$. In the worst case, each operation in $R(X)$ is checked against $G(X)$. The worst case time complexity is related to the worst case space complexity for a set of required operations. In the best case, Type-Check takes $O(1)$ because $R(X)$ is empty.

The time complexities for the type inference, adding/removing operations and renaming operations algorithms do not include the time complexity for type check. After each RO’s set of guaranteed operations or required operations is inferred, type check is performed if necessary. Most of the time type checking adds a constant factor of time to an algorithm’s time complexity. For example, for each RO affected by the algorithm Propagate-Requirements, type checking is performed. The time complexity of Propagate-Requirements now is $O(2 \times \text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$. The algorithms which are dominated by type checking include Axiom-GA’-add and Axiom-GA’-remove. The time complexity for each of these algorithms is multiplied by $O(\text{numVADTs} \times \text{numVADTROs})$. Table 4.15 summarizes the time complexity for Type-Check.
<table>
<thead>
<tr>
<th>User Action</th>
<th>Algorithm</th>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>formula edit, add/remove RO from VADT form, rename operation</td>
<td>Type-Check</td>
<td>O(numVADTs * numVADTROs)</td>
</tr>
</tbody>
</table>

Table 4.15. Worst case time complexity for the type-checking algorithm.

4.4.3 Discussion

In the previous Forms/3 type system [Burnett 1993], a Milner-style type system was supported. The algorithm for the previous system included unification and intersection. The time complexity was $O(msnA)$, where “$m$ is the number of sets being intersected for 1 operand, $s$ is the size of the smallest set being intersected for 1 operand, $n$ is the number of operands in the formula, and $A$ is the inverse Ackermann function, an extremely slow-growing function of $n$ ($A <= 5$ for $n <= 2^{65536}$)” [Burnett 1993]. In the previous type system, $m$ represented the number of ROs in the system, $s$ was the number of types or VADTs, and $n$ was the length of a formula. Using Core Forms/3 assumptions, $n$ and $A$ are set to $O(1)$, and the time complexity for the system is $O(ms)$.

A comparison between the two type systems is difficult because the previous system evaluated types on the granularity of names and accepted formulas that did not conform to our Core Forms/3 assumptions. A rough comparison, however, may be drawn. Translating the previous system into our notations, editing an RO’s formula in the previous system would require $O(numROs * numVADTs)$. In contrast, our system has at a time complexity of $O(numMatrixCells^2 * numVADTs * numVADTROs)$ because in the worst case, the system has to perform Axiom-GM on each affected matrix.
The intersection algorithm used in both systems contributes to each system's large time complexity. The intersection algorithm, however, is used for different purposes. In the previous system, intersection on the types of operands referenced in a formula is performed. In our system, intersection is performed on the guarantees of a matrix's plain cells. In a Forms/3 program, performance under our system depends on the number of matrices used. Under the previous system, performance depends on the number of operands and the number of sets associated with their inferred types.

Although the worst case time complexities appear to be large for some of our algorithms, these time complexities often are associated with matrix plain cells. Most of the algorithms that result in large time complexities involve inferring the guarantees or the requirements of matrices. For example, after a programmer edits RO X's formula, G(X) is propagated to all affected ROs V. Unless V is a matrix plain cell or an operation on a primitive form, no type inference is required. The type system only sets G(V) = G(X). If V is a matrix plain cell, then the set of guaranteed operations for V's parent matrix must be inferred using Axiom-GM. In programs without matrices, the time spent on type inference and type checking is reduced in comparison to programs with matrices.

For example, using simple cells instead of matrices can reduce the type checking time by a factor of four in the population study example (see Figure 2.5). In the example presented in Figure 2.5, if a programmer edited the simple cell city's formula to reference "100", the time complexity would be $O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$ where numMatrixCells = 4, numVADTs = 1 and numVADTROs = 23. If the programmer had used simple cells instead of the matrix graph, the time complexity would fall down to $O(\text{numROs} \times \text{numVADTs} \times \text{numVADTROs})$ where in this example numROs = 4.
Chapter 5: Future Research and Conclusion

5.1 Future Research

One important future step in the development of this type system is the design of a more suitable and effective type error interface. In our type system, type errors are displayed according to the production rules described by Djang [1998]. The production rules for displaying a type error message are [Djang 1998]:

\[
\begin{align*}
\text{errorMessage} &::= \text{cellID} \text{typeInformation} \text{typeInformation} \\
\text{typeInformation} &::= \text{primitiveOperations} \mid \text{operations} \\
\text{operations} &::= \text{operation} \mid \text{operation operations} \\
\text{operation} &::= \text{cellID} \mid \text{"Like" cellID}
\end{align*}
\]

Although these production rules support a minimally sized end-user vocabulary [Djang 1998], these type error messages still may not effectively communicate type errors to programmers and end-users. Future investigation in designing an effective type error interface may improve a user’s ability to use and understand a programming language.

One area of type error interface design is determining which type errors to display when type errors occur in several ROs after a formula edit. In the current implementation, a dialog box is displayed for each RO that contains a type error. This dialog box may be toggled off by the user. Research also should be conducted on if and how a type system can determine the source(s) of the type error. This research would be a contributing factor in designing an effective type error interface because it would influence decisions on the manner in which multiple type errors are displayed.

We also would like to learn whether or not users are able to effectively use the features supported by our type system. These features include the type error interface and similarity inheritance. We would like to determine whether or not
presenting type errors in terms of guarantees and requirements is effective in communicating type errors to users. We also would like to determine whether or not the concept of similarity inheritance is understandable and useful for users who are experienced with the copy and paste mechanisms of commercial spreadsheets.

Another area of research is a study on the performance of our type system. Empirical studies can be conducted on different categories of programs or on different types of subjects. The results should provide some estimations of expected space and time complexities for different types of programs or subjects.

Related to this study of performance is research on developing new algorithms or implementation devices which may reduce the time complexities for some of our algorithms. For example, in Propagate-Guarantees, the large time complexity arises when numMatrixCells are affected by a formula edit. This time complexity may be reduced using region information. Regions are collections of matrix plain cells that share the same formula. A matrix is composed of regions which are composed of matrix plain cells. The time complexity could be reduced if propagation to matrices does not occur until after all the matrix plain cells in a region have been updated.

Finally, during the implementation of our type system, some sections of our system were left incomplete or were not implemented. Most of these sections are dependent on other portions of Forms/3 that are not related to the type system. For example, primitive form operations are type checked by the underlying Lisp engine. A list of these incomplete parts are described in Appendix B.

5.2 Conclusion

Our type system supports implicit static typing. Our type system also supports inheritance without the introduction of explicit type declarations, a feature that is not found in any other statically typed declarative VPL. The advantages derived from our system include guaranteed program type safety (prior to program run-time), a small vocabulary of types and the support of inheritance including similarity inheritance.
The support of an implicit static type checking system removes the unnecessary programming mechanism of declaring types and allows VPLs to provide immediate visual feedback with respect to type errors. Using new data structures and algorithms to support the type inference and type checking axioms introduced in Djang et al.'s [1998], our type system can provide feedback on type errors without introducing the programmer or end-user to a large vocabulary. Type errors are reported with the RO's name and the operations guaranteed and required for that RO. The user's vocabulary is constrained to a small set of terms because the operations reported to the user are either operations the user created on a user-defined VADT form or operations the user will find on a primitive type form. The user is not introduced to any abstract type names. Another feature of the type system is the support of inheritance, including similarity inheritance. The support of similarity inheritance may be an important feature for a declarative VPL to support because it provides users with a powerful language-level code reuse programming mechanism.

In conclusion, this thesis has contributed an implementation of a model of types in a declarative VPL that provides programmers and end-users with programming features that were previously unavailable in other languages, and that also attempts to support these features without introducing the user to several new concepts. Our type system adheres to the type inference and type checking axioms presented in the Djang et al.'s model of types. In our research, we developed the data structures and algorithms to support the Djang et al.'s model of types and provided an implementation for these data structures and algorithms. In order to gain insight into our system's costs and performance, we also examined the space and time complexities for our data structures and algorithms.
Annotated Bibliography


Appendices
Appendix A: Additional Algorithms

A.1 Type Inference

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate-Requirements</td>
<td>Given: an RO X and an RO Z where Z is on a primitive form and X references Z</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve R(X).</td>
</tr>
<tr>
<td></td>
<td>2. Perform Axiom-R1a on Z and R(X).</td>
</tr>
<tr>
<td></td>
<td>3. No further propagation is necessary because Z is a result RO on a primitive form.</td>
</tr>
<tr>
<td></td>
<td>Given: a matrix X and a matrix Z that X references</td>
</tr>
<tr>
<td></td>
<td>1. If Z is on a VADT form, perform Axiom-R1bc-add.</td>
</tr>
<tr>
<td></td>
<td>2. Requirements do not need to be propagated for matrices.</td>
</tr>
</tbody>
</table>

Table A.1. Type inference algorithms.
<table>
<thead>
<tr>
<th>Propagate-Requirements-Rec</th>
<th>Given: an RO X and an abstraction box Z that X references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(X).</td>
<td></td>
</tr>
<tr>
<td>2. Perform Axiom-R1a on Z and R(X).</td>
<td></td>
</tr>
<tr>
<td>3. Retrieve the RO W referenced by Z.</td>
<td></td>
</tr>
<tr>
<td>4. Perform Propagate-Requirements-Rec on Z and W.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO X and a matrix plain cell Z that X references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(X).</td>
</tr>
<tr>
<td>2. Retrieve Z's parent matrix M.</td>
</tr>
<tr>
<td>3. Retrieve R(M).</td>
</tr>
<tr>
<td>5. Perform Union-Decr on R(M) and R(Z).</td>
</tr>
<tr>
<td>6. Perform Axiom-R1a on Z and R(X).</td>
</tr>
<tr>
<td>8. Perform Union-Incr on R(M) and R(Z).</td>
</tr>
<tr>
<td>9. Retrieve the RO W referenced by Z.</td>
</tr>
<tr>
<td>10. Perform Propagate-Requirements-Rec on Z and W.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO X and a simple cell Z that X references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(X).</td>
</tr>
<tr>
<td>2. Perform Axiom-R1a on Z and R(X).</td>
</tr>
<tr>
<td>3. Retrieve the RO W referenced by Z.</td>
</tr>
<tr>
<td>4. Perform Propagate-Requirements-Rec on Z and W.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO X and an RO Z where Z is on a primitive form and X references Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(X).</td>
</tr>
<tr>
<td>2. Perform Axiom-R1a on Z and R(X).</td>
</tr>
<tr>
<td>3. No further propagation is necessary because Z is a result RO on a primitive form.</td>
</tr>
</tbody>
</table>

Table A.1 (Continued).
<table>
<thead>
<tr>
<th>Undo-Ref</th>
<th>Given: a primitive constant Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements do not need to be undone</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix size cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements do not need to be un-unioned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix plain cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(Y). 2. If R(Y) = {}, then stop. 3. If Y is referenced by an RO (i.e., Y ← W), then perform Undo-Ref-Propagate on W and R(U). 4. Perform Union-Decr on R(Y) and R(U) and set list L equal to the list returned by Union-Decr. 5. Retrieve parent matrix M. 6. Perform Undo-Ref-Propagate-Matrix on M and L. 7. If Y is on a VADT form, then perform Undo-Ref-VADT.</td>
</tr>
</tbody>
</table>

Table A.1 (Continued).
<table>
<thead>
<tr>
<th>Undo-Ref-Propagate</th>
<th>Given: a simple cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(Y).</td>
<td>2. If R(Y) = {}, then stop.</td>
</tr>
<tr>
<td>3. If Y is referenced by another RO W (i.e., Y ← W), then perform Undo-Ref on W and R(U).</td>
<td>4. Perform Union-Decr on R(Y) and R(U).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a primitive constant Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements do not need to be undone</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix size cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Requirements do not need to be un-unioned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix plain cell Y that used to be referenced by X and R(U) where R(U) is a requirements set to be un-unioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve R(Y).</td>
</tr>
<tr>
<td>2. If R(Y) = {}, then stop.</td>
</tr>
<tr>
<td>3. If Y is referenced by an RO (i.e., Y ← W), then perform Undo-Ref-Propagate on W and R(U).</td>
</tr>
<tr>
<td>4. Perform Union-Decr on R(Y) and R(U) and set list L equal to the list returned by Union-Decr.</td>
</tr>
<tr>
<td>5. Retrieve parent matrix M.</td>
</tr>
<tr>
<td>6. Perform Undo-Ref-Propagate-Matrix on M and L.</td>
</tr>
</tbody>
</table>

Table A.1 (Continued).
### Table A.1 (Continued)

<table>
<thead>
<tr>
<th>Undo-Ref-Propagate-Matrix</th>
<th>Given: a matrix M and a list of operation names L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. If L is empty, then stop.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve R(M).</td>
</tr>
<tr>
<td></td>
<td>3. For each element op on L, perform the</td>
</tr>
<tr>
<td></td>
<td>following steps:</td>
</tr>
<tr>
<td></td>
<td>If op ∈ R(M), then retrieve op’s entry in</td>
</tr>
<tr>
<td></td>
<td>R(M), counter.</td>
</tr>
<tr>
<td></td>
<td>Decrement counter by 1.</td>
</tr>
<tr>
<td></td>
<td>If counter equals 0, then remove op from</td>
</tr>
<tr>
<td></td>
<td>R(M). Otherwise, place the new</td>
</tr>
<tr>
<td></td>
<td>counter value in op’s R(M) entry.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Union-Decr</th>
<th>Given: R(Y) and R(U) where R(U) contains the set of operations to be un-united from R(Y) and Y is a matrix plain cell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. For each operation in R(U), perform the following steps:</td>
</tr>
<tr>
<td></td>
<td>Find the operation in R(Y).</td>
</tr>
<tr>
<td></td>
<td>Retrieve the operation counter in R(Y).</td>
</tr>
<tr>
<td></td>
<td>Decrement the counter by 1.</td>
</tr>
<tr>
<td></td>
<td>If the counter equals 0, then remove the operation from R(Y) and place it on a</td>
</tr>
<tr>
<td></td>
<td>list L of removed operations.</td>
</tr>
<tr>
<td></td>
<td>Otherwise,</td>
</tr>
<tr>
<td></td>
<td>place the new counter in R(Y).</td>
</tr>
<tr>
<td></td>
<td>2. Return the list L.</td>
</tr>
<tr>
<td>Propagate-Guarantees-Rec</td>
<td>Given: an abstraction box V and an RO X that V references</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>1. No guarantees are propagated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO V on a primitive form and an RO X that V references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve G(V).</td>
</tr>
<tr>
<td>2. Retrieve G(X).</td>
</tr>
<tr>
<td>3. Set G(V) = G(X).</td>
</tr>
<tr>
<td>4. Retrieve the result RO W on V’s primitive form.</td>
</tr>
<tr>
<td>5. If G(W) needs to be derived, then derive G(W). Otherwise, stop.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix plain cell V and an RO X that V references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve G(V).</td>
</tr>
<tr>
<td>2. Retrieve G(X).</td>
</tr>
<tr>
<td>3. Set G(V) = G(X).</td>
</tr>
<tr>
<td>4. Retrieve parent matrix M.</td>
</tr>
<tr>
<td>5. Perform Axiom-GM on M.</td>
</tr>
<tr>
<td>6. Perform Propagate-Guarantees on M.</td>
</tr>
<tr>
<td>7. Perform Propagate-Guarantees on V.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: a matrix size cell V and an RO X that V references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve G(V).</td>
</tr>
<tr>
<td>2. Retrieve G(X).</td>
</tr>
<tr>
<td>3. Set G(V) = G(X).</td>
</tr>
<tr>
<td>4. Retrieve parent matrix M.</td>
</tr>
<tr>
<td>5. Perform Axiom-GM on M.</td>
</tr>
<tr>
<td>6. Perform Propagate-Guarantees on M.</td>
</tr>
<tr>
<td>7. Perform Propagate-Guarantees on V.</td>
</tr>
</tbody>
</table>

Table A.1 (Continued).
## A.2 Adding/Removing Operations

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate-Add-Guarantee-Operation</td>
<td>Given: an RO V and an RO X</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve the list of ROs L that reference V.</td>
</tr>
<tr>
<td></td>
<td>2. For each RO W on L, perform Propagate-Guarantee-Add-Operation-Rec on W and X.</td>
</tr>
<tr>
<td>Propagate-Add-Guarantee-Operation-Rec</td>
<td>Given: a matrix plain cell W that references V and an RO X</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve parent matrix M.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve a list of matrix plain cells L in M.</td>
</tr>
<tr>
<td></td>
<td>3. If X is a member of each cell’s guarantees, then add X to G(M) and perform Propagate-Guarantees on M.</td>
</tr>
<tr>
<td></td>
<td>4. Perform Propagate-Add-Guarantees on V and X.</td>
</tr>
<tr>
<td>Propagate-Remove-Guarantee-Operation</td>
<td>Given: an RO W that is not a matrix plain cell and an RO X</td>
</tr>
<tr>
<td></td>
<td>1. Perform Propagate-Add-Guarantees on W and X.</td>
</tr>
</tbody>
</table>

Table A.2. Algorithms for adding and removing operations.
Given: a matrix plain cell \( W \) that references \( V \) and an RO \( X \)

1. Retrieve parent matrix \( M \).
2. Retrieve \( G(M) \).
3. If \( X \) is a member of \( G(M) \), then remove \( X \) from \( G(M) \) and perform Propagate-Guarantees on \( M \).
4. Perform Propagate-Add-Guarantees on \( V \) and \( X \).

Given: an RO \( W \) that is not a matrix plain cell and an RO \( X \)

1. Perform Propagate-Add-Guarantees on \( W \) and \( X \).

Table A.2 (Continued).

### A.3 Renaming Operations

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate-NameChange-Guarantees</td>
<td>Given: an RO ( V ), an RO ( X ), an old name for ( X ) and a like name for ( X )</td>
</tr>
<tr>
<td></td>
<td>1. Retrieve a list ( L ) of ROs that reference ( V ).</td>
</tr>
<tr>
<td></td>
<td>2. For each RO ( W ) on ( L ), perform Propagate-NameChange-Guarantees-Rec.</td>
</tr>
</tbody>
</table>

Table A.3. Algorithms for renaming operations.
<table>
<thead>
<tr>
<th>Propagate-NameChange-Guarantees-Rec</th>
<th>Given: a matrix plain cell W, an RO X, an old name for X and a like name for X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve parent matrix M.</td>
<td>1. Retrieve parent matrix M.</td>
</tr>
<tr>
<td>3. Remove old name from G(M).</td>
<td>3. Remove old name from G(M).</td>
</tr>
<tr>
<td>4. Add X's new name to G(M).</td>
<td>4. Add X's new name to G(M).</td>
</tr>
<tr>
<td>5. If like name is used, then add like name to G(M).</td>
<td>5. If like name is used, then add like name to G(M).</td>
</tr>
<tr>
<td>6. Perform Propagate-NameChange-Guarantees on M.</td>
<td>6. Perform Propagate-NameChange-Guarantees on M.</td>
</tr>
<tr>
<td>7. Perform Propagate-NameChange-Guarantees on W, X, old name and like name.</td>
<td>7. Perform Propagate-NameChange-Guarantees on W, X, old name and like name.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO W that is on a primitive form, an RO X, an old name for X and a like name for X</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Remove old name from G(W).</td>
</tr>
<tr>
<td>3. Add X's new name to G(W).</td>
</tr>
<tr>
<td>4. If like name is used, add like name to G(W).</td>
</tr>
<tr>
<td>5. Perform Propagate-NameChange-Guarantees on W, X, old name and like name.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Given: an RO W that is not a matrix plain cell and is not on a primitive form, an RO X, an old name for X and a like name for X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Perform Propagate-NameChange-Guarantees on W, X, old name and like name.</td>
</tr>
</tbody>
</table>

Table A.3 (Continued).
Propagate-NameChange-Requirements

<table>
<thead>
<tr>
<th>Given: an RO Y, an RO X and an old name for X where Y references an RO Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieve the RO Z that Y references.</td>
</tr>
<tr>
<td>4. Retrieve the counter for old name in R(Z).</td>
</tr>
<tr>
<td>5. Remove old name from R(Z).</td>
</tr>
<tr>
<td>6. Add X’s new name to R(Z) with the counter.</td>
</tr>
<tr>
<td>7. If Z references a constant or an abstraction box, then stop.</td>
</tr>
<tr>
<td>8. Perform Propagate-NameChange-Requirements on Z, X and old name.</td>
</tr>
</tbody>
</table>

Table A.3 (Continued).
### A.4 Time Analyses

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate-Requirements-Rec</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$ where $\text{numMatrixCells} = \frac{(\text{numROs}/4) - 1}{\text{numROs} - 4}$</td>
</tr>
<tr>
<td>Undo-Ref</td>
<td>$O(\text{numMatrixCells} \times \text{numVADTs} \times \text{numVADTROs})$ where $\text{numMatrixCells} = \text{numROs} - 3$</td>
</tr>
<tr>
<td>Undo-Ref-Propagate-Matrix</td>
<td>$O(\text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>Union-Decr</td>
<td>$O(\text{numVADTs} \times \text{numVADTROs})$</td>
</tr>
<tr>
<td>Propagate-Guarantees-Rec</td>
<td>$O(\text{numMatrixCells}^2 \times \text{numVADTs} \times \text{numVADTROs})$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-Add-Guarantee-Operation</td>
<td>$O(\text{numMatrixCells}^2)$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-Add-Guarantee-Operation-Rec</td>
<td>$O(\text{numMatrixCells}^2)$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-Remove-Guarantee-Operation</td>
<td>$O(\text{numMatrixCells})$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-Remove-Guarantee-Operation-Rec</td>
<td>$O(\text{numMatrixCells})$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-NameChange-Guarantees</td>
<td>$O(\text{numMatrixCells})$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-NameChange-Guarantees-Rec</td>
<td>$O(\text{numMatrixCells})$ where $\text{numMatrixCells} = \text{numROs} - 4$</td>
</tr>
<tr>
<td>Propagate-NameChange-Requirements</td>
<td>$O(\text{numROs})$</td>
</tr>
</tbody>
</table>

Table A.4._summary of worst case time complexities for additional type system algorithms.
Appendix B: Source Code Examples

B.1 types3.lisp

The file types3.lisp contains functions that support the type system implemented in this thesis. The following are some of the functions resulting from the type inference and type checking algorithms presented in this thesis.

B.2 type-check

;;; -------------------------------
;;; type-check (RO)
;;;
;;; Type inference and type checking occurs after
;;; the formula is changed in order to be
;;; consistent with other Forms/3 features. For
;;; example, create a form X and a cell x. If you
;;; give the cell x the formula "y", *NO-SUCH-CELL*
;;; error occurs, but the formula remains "y".
;;;
;;; Precondition(s):
;;; 1) The formula in anRO has already been set
;;; to the new formula.
;;; 2) Assume Core Forms/3 and basic formula
;;; model. That is, the only operator is the
;;; "reference" operator.
;;; Given: anRO an RO
;;; anOldFmla
;;; keyword
;;; eval
;;; Returns:
;;;
;;; (defmethod type-check ((anRO RO) anOldFmla &key
;;; (eval nil))
;;;)
;;; Set the old formula and the old required set
;;; of operations
;;; (let ((oldFmla (if (stringp anOldFmla)
;;; (read-from-string anOldFmla)

...
(anOldFmla)
(oldReqSet (displayable-requiredOps anRO))

;; If the formula is the same as before,
;; no type-checking needs to be done
(if (or eval (not (equalp
(displayable-formula anRO)
oldFmla)))

(progn

;; Clear out the previous formula by
;; undoing unions
(type-undo-ref anRO oldFmla
 :requiredSet oldReqSet)

;; Type-check anRO and set its new
;; sets of ops. Then, type-check
;; and propagate the changes to all
;; direct and indirect deps
(type-check-RO anRO)
(type-check-RO-deps anRO)

) ; End progn

) ; End if

) ; End let

) ; End type-check (RO)
B.3 type-undo-ref

;;; -------------------------------
;;; type-undo-ref
;;; type-undo-ref replaces undo-union by
decrementing counters. Calls type-union-decr
to actually undo union. type-undo-ref
determines the dependencies that need to have
their sets un-unioned.

;;; Precondition(s):
Given:
  anRO
  anOldFmla
:requiredSet

Returns:

(defun type-undo-ref (anRO anOldFmla &key (requiredSet nil))

;;; Retrieve the affectingRO and the
;;; appropriate set of required ops to undo
(let ((affectingRO
  (type-whoAffectsMe anRO :fmla anOldFmla :useFmla t))
  (oldReqOps
    (if requiredSet requiredSet
      (displayable-requiredOps
        anRO)))))

;;; If nobody affects me, then we're done
;;; Catches constants
(if affectingRO
  (progn
    ;; If the formula is an RO ref, then
    ;; propagate requirements. Otherwise,
    ;; perform a type check on the primitive
    ;; form operation with its new
    ;; set of requirements
    (if (typep affectingRO 'RO)
(progn

;; If oldReqOps is an empty hash
;; table, then no undo-refs needed
(if (not (zerop (hash-table-count oldReqOps)))
  ;; Undo indirect dependencies
  (type-undo-ref-propagate
   affectingRO :requiredSet oldReqOps))

;; Check if the RO is on a VADT
;; form. If the affectingRO is on a
;; VADT form, check if it still
;; belongs in the requirements set
(if (displayable-vadt-form-p (displayable-parentForm affectingRO))
  (type-remove-requirements-operation affectingRO anRO)))

;; If the affectingRO has a formula,
;; then we don't handle it now
(type-check-primitiveFormOperation affectingRO anOldFmla)

);) ; End if affectingRO is RO

));) ; End if affectingRO is nil

);) ; End let

);) ; End type-undo-ref
116

B.4 type-check-RO

;;; ---------------------------------------------
;;; type-check-RO (RO)
;;;
;;; type-check-RO determines the appropriate axiom
;;; to call, sets the new sets of ops, and then
;;; performs a type-check. If an error occurs,
;;; then type-check-fail is called.
;;;
;;; Precondition(s):
;;; Given: anRO
;;; Returns:
;;;
(defun type-check-RO ((anRO RO)
  &key (empty nil))

;;; Determine the appropriate axiom to use and
;;; retrieve the new guarantees
(let* ((emptyFmla (if (or empty
  (null (displayable-formula anRO)))
  t nil))
  (newGuarOps (if emptyFmla
    (make-init-guaranteedOps)
    (axiom-GC-or-Gref anRO)))
  (parentForm (displayable-parentForm
    anRO)))

;;; Set the new guar ops
(setf (displayable-guaranteedOps anRO)
  newGuarOps)

;;; Type-check
(if (type-is-r-a-subset-of-g anRO)

;;; If a type error existed previously,
;;; remove it
(if (displayable-typeError anRO)
  (type-remove-typeError anRO))

;;; Otherwise set the type error
(type-check-error anRO))

;; If the RO is on a copied VADT form, then
;; this RO's new fmla implies that it has been
;; added to the form's defSet. Make sure anRO
;; is not in an absbox or in a matrix.
(if (and (displayable-vadt-form-p parentForm)
         (not (displayable-modelp parentForm))
         (not (displayable-inAbsBox anRO))
         (type-add-requirements-operation anRO))

;; Also if this RO is on a copied VADT form,
;; then we have to make a copy of the main
;; absbox's guarantees because it no longer
;; shares with the original type. Also remove
;; like ops.
(if (and (displayable-vadt-form-p parentForm)
         (not (displayable-modelp parentForm))
         (not (displayable-inAbsBox anRO)))
  (progn
    (type-copy-guarantee-set
     (displayable-absBox parentForm))
    (type-remove-like-operation anRO)
    (type-check-guaranteedDeps
     (displayable-absBox parentForm)))))

) ; End let

) ; End type-check-RO (RO)
B.5 type-check-RO-deps

;;; -------------------------------
;;; type-check-RO-deps
;;;
;;; type-check-RO-deps propagates an RO's formula's effects to all direct and indirect dependencies. These dependencies include those affected by it and those that affect it. For those affected by it (whoDoIAffect), guaranteed sets of operations are affected. For those that affect it (whoAffectsMe), required sets of operations are affected.
;;;
;;; Precondition(s):
;;; Given: anRO
;;; Returns:
;;;

(defun type-check-RO-deps (anRO)

    ;; Since an RO is first affected before it can affect, we propagate changes downstream first. Change all affected required sets of operations
    (type-check-reqDeps anRO)

    ;; Propagate changes upstream; change all affected guaranteed ops
    (type-check-guaranteedDeps anRO)
)

); End type-check-RO-deps
B.6 type-check-requiredDeps

;; type-check-reqDeps (RO)
;;
;; Type-checking is performed in this direction
;; because these ROs may have a new set of
;; required ops which may or may not be a subset
;; of their guaranteed ops.
;;
;; Given: anRO
;; Returns: a boolean indicating whether or not
;; the direct and indirect formula
;; deps of anRO are type safe
;;
(defmethod type-check-reqDeps ((anRO RO))

(let ((affectingRO (type-whoAffectsMe anRO))
    (requiredSet (displayable-requiredOps anRO)))

    ;; If no one affects anRO, then we're done.
    (if affectingRO

        ;; If the affectingRO is an RO reference
        (cond ((typep affectingRO 'RO)
            (type-propagate-requirements anRO requiredSet))

        ;; If the affectingRO is on a
        ;; primitive form
        ((is-a-primitiveFormOperation-fmla anRO)
            (type-check-primitiveFormOperation anRO
                nil))

        ;; Otherwise, it's unidentified or a
        ;; Lisp function
        (t
            (format t
                "Unidentified affectingRO"))))
B.7 type-check-guaranteedDeps

;;; ___________________________________________________________
;;; type-check-guaranteedDeps (RO)
;;; type-check-guaranteedDeps propagates the new
;;; set of guaranteed ops to all affected ROs.
;;; Given: anRO
;;; Returns:

(defun type-check-guaranteedDeps ((anRO RO))

;;; Formula and copy dependencies are treated
;;; separately
(let* ((guaranteedOpsSet (displayable-guaranteedOps anRO))
       (fmlaList (WAWTable-find :formula-dep
                              (make-cellRef
                               (displayable-parentFormID anRO)
                               (displayable-id anRO))))
       (copyList (WAWTable-find :copy-dep
                                (make-cellRef
                                 (displayable-parentFormID anRO)
                                 (displayable-id anRO))))
       ;; If anRO affects any RO, then propagate
       ;; guaranteed set of ops
       (if fmlaList
           ;; For each dep, perform type-is-r-a-subset-
           ;; of-g on its new guaranteed ops set. Also
           ;; set the new guaranteed ops set
...
(mapcar #'(lambda (affectedRO)

  (let* ((thisRO (cellRef-cell affectedRO))
         (parentForm
           (displayable-parentForm
            thisRO))
         (VADTForm?
           (displayable-vadt-form-p
            parentForm)))
    ;; Make sure this isn't a region, a ;; matrix, the same absbox as anRO, ;; and isn't the image cell's ;; absbox
    (if (and (not (displayable-isRegion? thisRO))
            (not (typep thisRO 'matrix))
            (not
             (equalp
              (displayable-inAbsBox
               anRO)
              thisRO))
            (not (equalp thisRO
                         (if VADTForm?
                             (displayable-absBox
                              parentForm)
                             nil)))))
    ;; Handle special case for ;; primitive form ops
    (if (is-a-primitiveFormOperation-fmla thisRO)
        (type-check-primitiveFormOperation
            anRO (displayable-formula
                  thisRO))
        (progn
            (setf (displayable-guaranteedOps thisRO) guaranteedOpsSet))
    ;; If a type error occurs,
;;; throw an error msg
(if (type-is-r-a-subset-of-g thisRO)
  (if (displayable-typeError thisRO)
    (type-remove-typeError thisRO))
  (type-check-error thisRO))

;;; Recursively propagate to dependencies
(type-check-guaranteedDeps thisRO)))))
)
)
); End let
)
); End lambda
fmlaList)
)
); End if

;;; We handle copied ROs differently. If the similarity is both name and formula or just formula, then continue propagation. Otherwise, we leave the RO alone and don't propagate
(if copyList
  (mapcar "'(lambda (thisCellRef)
    (let* ((thisRO (cellRef-cell thisCellRef))
      (thisParentForm (displayable-parentForm thisRO))
      (similarity (displayable-similarity thisRO)))
    (if (or
      (equalp similarity 'copy)
      (equalp similarity 'formula-only))
      (progn
      (fmlaList))))))
(setf (displayable-guaranteedOps thisRO) guaranteedOpsSet)

;; If a type error
;; occurs, throw an error
;; msg
(if (type-is-r-a-subset-of-g thisRO)
  (if (displayable-typeError thisRO)
      (type-remove-typeError thisRO)
      (type-check-error thisRO))

(type-check-error thisRO))

;; Recursively propagate to
;; dependencies
(type-check-guaranteedDeps thisRO)))

)); End let and lambda

copyList)); End copyList

)); End let

); End type-check-guaranteedDeps (RO)
B.8 axiom-GC

;;;; axiomatic-GC
;;;;
;;;; axiom-GC conforms to the axiom [GC] described
;;;; in Chapter 5 of Djang's dissertation: Given
;;;; X<--C where C is a constant, G(X)=G(C) where
;;;; G(C)={y|y is an element in C's ROset}.
;;;;
;;;; Given: aCell
;;;; Returns: a hash table of guaranteed
;;;; ops
;;;;
(defun axiom-GC (aCell &key (fmla nil))

;;;; Using the constant in aCell's formula, access
;;;; its :guaranteedOps slot. We have to copy the
;;;; set of ops because we don't want to change the
;;;; operations accidentally.
(if fmla
    (type-copy-primitiveTypeOperations
     (is-a-constant-fmla aCell :aFmla fmla))
    (type-copy-primitiveTypeOperations
     (is-a-constant-fmla aCell)))

);; End axiom-GC
B.9 axiom-GM

;;; axiom-GM

;;; axiom-GM conforms to the axiom [GM] described
;;; in Chapter 5 of Djang's dissertation: Given
;;; matrix M, \( G(M) = (the \ intersection \ of) \ G(M[i]) \)
;;; for all \( M[i] \) is an element of \( M.gridROset \), i.e.
;;; for all cells in matrix M.

;;; Given: aMatrix
;;; Returns: a hash table of guaranteed ops

(defun axiom-GM (aMatrix)
  ;; Take the intersection of all guaranteed ops
  ;; of the cells in aMatrix. If no matrix
  ;; plain cells exist, then the set of
  ;; guaranteed ops is empty.
  (if (> (length (displayable-cellList aMatrix)) 2)
      (type-matrix-intersection aMatrix)
    (make-init-guaranteedOps))
)

);; End axiom-GM
B.10 axiom-GA'prime-add

;;; -----------------------------------------------
;;; axiom-GA'prime-add
;;; axiom-GA'prime-add conforms to the axiom [GA']
;;; described in Chapter 5 of Djang's dissertation:
;;; Given an absbox A, G(F:A)={x, like y | x is an
;;; element of F.ROset, y --> x}. axiom-GA is not
;;; used because axiom-GA'prime-add and axiom-
;;; GA'prime-add replace it. axiom-GA'prime-add is
;;; called when cells are added to a VADT form.
;;; Precondition(s):
;;; Given: anRO
;;; Returns:

(defun axiom-GA'prime-add (anRO)
  ;; Retrieve VADT form and absbox info
  (let* ((VADTForm (displayable-parentForm anRO))
         (mainAbsBox (displayable-absBox VADTForm))
         (guaranteedOpsSetAbsBox
          (displayable-guaranteedOps mainAbsBox))
         (opName (type-make-opName VADTForm anRO)))
    (if (not (gethash opName guaranteedOpsSetAbsBox))
      (setf (gethash opName guaranteedOpsSetAbsBox) opName)
    )
  )
) ; End axiom-GA'prime-add
B.11 axiom-GA'prime-remove

;;; -----------------------------------
;;; axiom-GA'prime-remove
;;; axiom-GA'prime-remove conforms to the axiom
;;; [GA'] described in Chapter 5 of Djang's
dissertation: Given an absbox A, \( G(F:A) = \{ x, \] like y | x is an element of F.ROset, y \rightarrow x \}.

axiom-GA is not used because axiom-GA'prime-add
and axiom-GA'prime-remove replace it. axiom-
GA'prime-remove is called when cells are removed
from a VADT form.

;;;

;;; Precondition(s):

;;; Given: anRO

;;; Returns:

(defun axiom-GA'prime-remove (anRO)

;;; Retrieve VADT form and absbox info
(let* ((VADTForm (displayable-parentForm anRO)))
  (mainAbsBox (displayable-absBox VADTForm))
  (guaranteedOpsSetAbsBox
   (displayable-guaranteedOps mainAbsBox))
  (opName (type-make-opName VADTForm anRO)))

(if (gethash opName guaranteedOpsSetAbsBox)
  (remhash opName guaranteedOpsSetAbsBox)
 ) ; End if

) ; End axiom-GA'prime-remove
B.12 axiom-R1a

;;; ------------------------------------------
;;;  axiom-R1a
;;;  
;;;  axiom-R1a conforms to the first part of axiom
;;;  [R1] described in Chapter 5 of Djang's
dissertation:
;;;  
;;;  Given the formula:
;;;  
X1, X2, ... Xn ← Y and Z1 ←
;;;  Fy(defSet1):Op1, Z2 ← Fy(defSet2):Op2,
;;;  ...Zm ← Fy(defSetm):Opm
;;;  
Then R(Y) =
;;;  (Union of) i=1..n of R(Xi) (unioned
;;;  with) {Op1,Op2,...Opm} (unioned
;;;  with) (Op | Op is an element of
;;;  defSetk)

;;; Precondition: 1) Assumes Core Forms/3 and
;;; basic formula model
;;; Given: anRO
;;; Returns:
;;;
(defun axiom-R1a (anRO aRequiredSetToAdd)

(let ((newReqOps (displayable-requiredOps anRO))
       (addedOpsHT (make-hash-table)))

  ;; If the new required ops table is not empty,
  ;; union-incr
  (if (not (zerop (hash-table-count
                   aRequiredSetToAdd)))
    (setf addedOpsHT (type-union-incr
                      newReqOps
                      aRequiredSetToAdd))

  ;; Return the added operations
  addedOpsHT)

) ; End let
) ; End axiom-R1a
B.13 axiom-R1bc-add

;;; ----------------------------------------
;;; axiom-R1bc-add
;;; 
;;; axiom-R1bc-add conforms to the second part of
;;; axiom [R1] described in Chapter 5 of Djang’s
;;; dissertation.
;;; 
;;; Precondition(s): 1. anRO is on a VADT form.
;;; Given: anRO
;;; Returns:
;;; 
(defun axiom-R1bc-add (anRO)

;;; Retrieve the VADT form, the absbox, the name
;;; of anRO and the required set of ops for the
;;; absbox
(let* ((VADTForm (displayable-parentForm anRO))
       (mainAbsBox (displayable-absBox
                    VADTForm))
       (opName (type-make-opName VADTForm anRO))
       (refdRO (type-whoAffectsMe mainAbsBox))
       (reqSet (if refdRO (displayable-
                        requiredOps refdRO))))

(if refdRO
  (if (gethash opName reqSet)
    (setf (gethash opName reqSet)
          (+ 1 (gethash opName reqSet)))
    (setf (gethash opName reqSet) 1)))

) ; End let*

) ; End axiom-R1bc-add
B.14 axiom-R1bc-remove

;; type-remove-requirements-operation (RO)
;; type-remove-requirements-operation removes a
;; new operation to an RO's set of requirements.
;; Also propagates this information to all
;; affected ROs.

;; Precondition(s):
;; Given: anRO
;; Returns: anAffectedRO

(defun type-remove-requirements-operation
  (anRO RO) anAffectedRO)

;; Retrieve the VADT form, the absbox, the
;; name of anRO and the required set of ops
;; for the absbox
(let* ((VADTForm (displayable-parentForm anRO))
   (mainAbsBox (displayable-absBox VADTForm))
   (opName (type-make-opName VADTForm anRO))
   (refdRO (type-whoAffectsMe mainAbsBox))
   (reqSet (if refdRO (displayable-requiredOps refdRO))
     (ht (make-hash-table)))
   (if (type-is-x-in-defSet? anRO)
     (progn (setf (gethash opName ht) 1)
      (type-undo-ref-propagate refdRO :requiredSet ht))
     (if (and refdRO
      (not (equalp (displayable-parentForm anAffectedRO)))))}
(displayable-parentForm anRO)))
(progn
  (setf (gethash opName ht) 1)
  (type-undo-ref-propagate refdRO
   :requiredSet ht)))
)
; End let*
)
; End type-remove-requirements-operation

B.15 axiom-RM

;;; -----------------------------------------------
;;; axiom-RM
;;; axiom-RM conforms to the first part of axiom
;;; [RM] described in Chapter 5 of Djang's dissertation.
;;; Precondition(s):
;;; Given: aMatrix
;;; Returns: a hash table of required ops
;;; (defun axiom-RM (aMatrix)

;;; Take the intersection of all guaranteed ops
;;; of the cells in aMatrix. If the matrix
;;; contains no cells other than NumRows and
;;; NumCols, then return nil
(cond ((<= (length (displayable-cellList aMatrix)) 2)
   (make-init-requiredOps))
  (t (type-matrix-union aMatrix))
)
; End cond
)
; End axiom-RM
B.16 axiom-RN

;;; --- ------------------------------------------------------
;;; axiom-RN
;;;
;;; Called when a matrix is first created. The set
;;; of required operations can never change for
;;; NumRows and NumCols.
;;;
;;; Given: aCell
;;; Returns:
;;;
;;; (defun axiom-RN (aCell)

;;; Place numberOps in the hash table.
;;; (let ((numberOpsSet (gethash 'number
;;;   $PrimitiveTypeOperations»
;;;   (rx (make-init-requiredOps»)

;;; Initialize the counter to 1 for each op in
;;; rx
;;; (maphash #'(lambda (key value) (setf (gethash
;;; key rx) 1))
;;; numberOpsSet)

;;; Return the hash table
;;; rx

} ; End let
)
);

) ; End axiom-RN

B.17 Implementation Details

Forms/3 is a research language in which some portions of the language are
under development. The following is a list of incomplete or missing portions of our
type system. The list is presented for current and future Forms/3 implementors.

- Loading forms do not initialize the ROs’ sets of operations.
• Impersonation errors are handled by generalization. Whether or not they should be handled by the type system and in what manner they should be handled has not been decided.

• Copied ROs do not have their sets of operations properly updated in the following situation: given ROs X and Y and their respective copies X' and Y', if Y ← X such that Y' ← X', then the set of operations for Y' is shared with Y and is not independent. The result is that if a user changes the formula for X', the sets of operations for Y' still reflects the sets shared with Y.

• Primitive forms have not been implemented as well as the algorithms to handle them. The current implementation only retrieves a hash table of operations from $\text{PrimitiveFormsOpsG}$ or $\text{PrimitiveFormsOpsR}$. For primitive forms such as "if" that require intersection or union algorithms, these forms are not handled. The reason is that some of these operations are handled by the Lisp engine (see operators3.lisp).

• A more appropriate and effective type error interface should be developed. The current interface consists of a dialog box displaying the RO's name, and the type information is displayed in the Lisp interpreter.