An Abstract Of The Dissertation Of

Rebecca Djang for the degree of Doctor of Philosophy in Computer Science presented on December 17, 1998. Title: Similarity Inheritance: A Model of Inheritance for Declarative Visual Programming Languages.

Abstract approved: _____

Margaret M. Burnett

Declarative visual programming languages (VPLs), including spreadsheets, make up a large portion of both research and commercial VPLs. Spreadsheets in particular enjoy a wide audience, including end users. Unfortunately, spreadsheets and most other declarative VPLs still suffer from some of the problems that have been solved in other languages, such as ad-hoc (cut-and-paste) reuse of code which has been remedied in object-oriented languages, for example, through the code-reuse mechanism of inheritance. We believe spreadsheets and other declarative VPLs can benefit from the addition of an inheritance-like mechanism for fine-grained code reuse. This dissertation first examines the opportunities for supporting reuse inherent in declarative VPLs, and then introduces *similarity inheritance* and describes a prototype of this model in the research spreadsheet language Forms/3. Similarity inheritance is very flexible, allowing multiple granularities of code sharing and even mutual inheritance; it includes explicit representations of inherited code and all sharing relationships, and it subsumes the current spreadsheet mechanisms for formula propagation, providing a gradual migration from simple formula reuse to more sophisticated uses of inheritance among objects. Since the inheritance model separates inheritance from types, we investigate what notion of types is appropriate to support reuse of functions on different types (operation polymorphism). Because it is important to us that immediate feedback, which is characteristic of many VPLs, be preserved, including feedback with respect to type errors, we introduce a model of types suitable for static type inference in the presence of operation polymorphism with similarity inheritance.

Similarity Inheritance:
A Model of Inheritance for Declarative Visual Programming Languages

by

Rebecca Djang

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed December 17, 1998
Commencement June 1999

Doctor of Philosophy dissertation of Rebecca Djang presented on December 17, 1998.

APPROVED:

Major Professor, representing Computer Science

Chair of Department of Computer Science

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for privacy

Rebecca Djang, Author

Acknowledgments

First, I must express my deepest gratitude to my advisor Dr. Margaret Burnett whose guidance made it possible for me to complete this dissertation. I am thankful for all the time she devoted to helping me move forward with my research and writing. I especially appreciate her patience, optimism and encouragement at the times when I was most ready to give up. I am blessed to have worked with such an excellent mentor.

I would like to express my appreciation to the present and former members of the Forms/3 research group for their dedicated implementation efforts upon which my work builds and also for their helpful comments and suggestions. These members include Sherry Yang, John Atwood, Paul Carlson, Pieter van Zee, Maureen Cheshire, Herky Gottfried, Judy Hays, Anurag Agrawal, JJ Cadiz, Eric Wilcox, Lixin Li, Sunanda Mishra, Chris DuPuis, David Hackenyos, Dusty Reichwein, Roger Chen, Andrei Sheretov, Zach Welch, Karen Rothermel, Tim Adams and Frank Cort.

I thank Drs. Timothy Budd, Curtis Cook, Gregg Rothermel and David Hann for serving on my committee and for their helpful comments on the dissertation.

I am also grateful to Hewlett-Packard, the National Science Foundation, and the NASA Graduate Student Researchers Program for their financial support of this research.

A very special thank you to Sarah Deel and Nancy Silva—the original coffee group—and their successor Annabelle van Zyl. These women were invaluable in keeping my struggles in perspective as they shared their own stories along the way. Other members of the Graduate Christian Forum in 1997 and 1998 also provided valuable emotional support.

Finally, I am deeply grateful to my parents for their never-ending support and to my husband Kevin for his patience, understanding and encouragement throughout this process.

Table Of Contents

Table Of Contents (Continued)

Table Of Contents (Continued)

List Of Figures

List Of Tables

Dedication

To my godfather Bob Blucher who delighted in sharing new puzzles and games with me and encouraged my knowledge of computer programming, and to my sixth grade math and computers teacher Mr. Forvilly who taught me that math and computer programming are great games.

Similarity Inheritance:
A Model of Inheritance for Declarative Visual Programming Languages

Rebecca Djang

Department of Computer Science

College of Engineering

Oregon State University

Similarity Inheritance: A Model of Inheritance for Declarative Visual Programming Languages

Chapter 1: Introduction

We are interested in advancing the expressive power of visual programming languages (VPLs), especially those that are both declarative and responsive. By declarative we mean that programming is a matter of specifying the relationships among data in the program. (This is in contrast to languages in which data is changed directly or by sending messages to objects to modify their state.) By responsive, we mean that whenever any new piece of information enters the system—such as when the system computes new values or the programmer edits the program—the effects are immediately and automatically reflected in the displayed portion of the program's results.

This research has been prototyped using a spreadsheet language. Spreadsheets are not only declarative and responsive, they are also used by a wide audience, including end users. We believe some of the powerful features of modern programming languages could benefit end users as well as programmers as long as the features are provided by mechanisms that allow for gradual migration from very simple use to sophisticated use. Spreadsheets have proven to be a popular programming paradigm, accessible even to non-programmers. Current spreadsheets, however, suffer from some of the problems that have been solved in other programming languages. For example, in other programming languages, object-oriented inheritance mechanisms have improved upon ad-hoc (cut-and-paste) reuse of code, but spreadsheets still support only ad-hoc reuse through copy/paste and formula replication. Thus spreadsheet users must remember the reuse relationships themselves and maintain them manually whenever a reused formula changes. Some commercial spreadsheets such as Excel® have a few additional conveniences, such as automated formula adjustment when a new copy of a linked

spreadsheet is made. However, these features are simply editing conveniences, and the user is still left to manually maintain the reuse relationships.

Incorporating inheritance into declarative VPLs could result in stronger support for formula reuse than is found in current spreadsheets. However, existing models of inheritance do not seem suitable for the spreadsheet paradigm because they introduce concepts foreign to spreadsheets, such as message passing. The main goal of this research was to find an approach to inheritance-like code reuse suitable for declarative VPLs such as spreadsheet VPLs.

We use the term *spreadsheet VPLs* to refer to a variety of systems that follow the spreadsheet paradigm, from commercial spreadsheets to more sophisticated research VPLs that follow the declarative, one-way constraint evaluation model. The essence of the paradigm is summarized by Alan Kay's *value rule* for spreadsheets [Kay 1984], which states that a cell's value is defined solely by the declarative formula explicitly given it by the user. In addition, as mentioned earlier, spreadsheets are responsive. We clarify the definition of responsiveness in terms of Tanimoto's liveness scale [Tanimoto 1990]. *Liveness* describes the amount and availability of semantic feedback provided to the programmer. At level 1, no feedback is provided. At level 2, the programmer can receive semantic feedback on request (such as from an interpreter). 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 other events such as system clock ticks. *Responsive* refers to level 3 or greater liveness.

In this dissertation we present a new approach to inheritance suitable for declarative VPLs, and an instantiation of the approach in the research spreadsheet VPL Forms/3 [Atwood et al. 1996; Burnett and Gottfried 1998]. The approach, called

similarity inheritance, provides a concrete way of sharing behavior among objects in a declarative, responsive VPL. The unique attributes of similarity inheritance are that it:

- is flexible enough to allow sharing at multiple granularities and even allows mutual inheritance;
- includes an explicit visual representation of all the object's unique and shared behaviors, rather than leaving some behaviors implied through parenthood;
- subsumes the current spreadsheet edit-based mechanisms for formula propagation, unifying formula reuse with inheritance;
- brings object-oriented concepts to declarative VPLs without using external languages or macros.

1.1 Inheritance and Reuse

Inheritance as a language feature is one way to enable reuse; however, the literature uses the term reuse in many contexts. The majority of uses of the term fall into the following categories:

1. Code sharing: the structured programming technique of encapsulating procedures for use from different parts of a program is widely used in almost all programming languages. In object-oriented languages, code sharing is usually realized through inheritance, which allows methods (procedures) to be shared by many objects. Polymorphism allows code to be shared over a wide range of types.
2. Code components in repositories: pieces of code are collected in a library where they are available for reuse.
3. Design and other noncode artifacts: design patterns, specification reuse, and reuse of any other product of the software development process.
4. Organizational or management issues: nontechnical ways to facilitate and promote reuse in the workplace.

While the context for similarity inheritance is the first category (code sharing), we will also explore opportunities for supporting reuse already present in declarative VPLs (such as spreadsheet VPLs) in the context of the second category (code repositories). Additionally, we will spend some time discussing the implications of similarity

inheritance for static typing and, specifically, polymorphism. Thus reuse provides a context for all of the research described.

1.2 Introduction to Forms/3

The ideas in this dissertation have been prototyped in the research spreadsheet language Forms/3 [Atwood et al. 1996; Burnett and Gottfried 1998]. This section provides the necessary background in Forms/3 to understand the examples in later chapters.

A Forms/3 programmer creates a program by using direct manipulation to place cells on forms (spreadsheets) and to define a formula for each cell using a flexible combination of pointing, typing, and gesturing. A program's calculations are entirely determined by these formulas (see Table 1.1). The formulas combine into a network of one-way constraints, and the system continuously ensures that all values displayed on the screen satisfy these constraints.

formula	::=	*BLANK* | expr
expr	::=	*CONSTANT* | ref | infixExpr | prefixExpr | ifExpr
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	unaryPrefixOperator subExpr | binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr |
		IF subExpr THEN subExpr
subExpr	::=	*CONSTANT* | ref | (expr)
infixOperator	::=	+ | - | * | / | AND | OR | = | ...
unaryPrefixOperator	::=	ROUND | CIRCLE | ...
binaryPrefixOperator	::=	APPEND | ...
ref	::=	*CELL* | *MATRIX* | *ABS* | *ABS* [*CELL*] | *MATRIX* [subscripts] | *ABS* [*MATRIX*] |
		ABS [*MATRIX*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

Table 1.1 A subset of the grammar for the formula language used in this dissertation. Not included in this subset are the complete set of operators and 6 "pseudo references"—I, J, NUMROWS, NUMCOLS, LASTROW, and LASTCOL—which are used in matrix subscripts.

As this brief description shows, like other spreadsheet languages, Forms/3 is declarative and responsive. One of our goals was to maintain these two features, incorporating them into the support for reuse.

Even before the addition of similarity inheritance, Forms/3 has long supported an extensible collection of types. Attributes of a type are defined by formulas in the cells on a type definition form (called a visual abstract data type form, or VADT form). An instance of a type is the value of a cell, which can be referenced just like any other cell. For example, the built-in circle type shown in Figure 1.1 is defined by cells defining its radius, line thickness, color, and other attributes. One way to instantiate a circle is to copy the circle form, changing any formulas necessary to achieve the desired attributes (as in the figure); another way is to graphically define its attributes [Gottfried and Burnett 1997], such as by sketching a new circle or by stretching an existing circle by direct manipulation. The graphical way is a shortcut for the first way, and we will use only the first way in this dissertation.

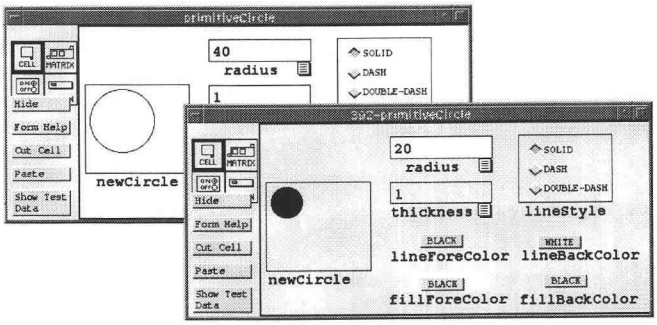

Figure 1.1 The built-in circle type. The white form primitiveCircle is a built-in form that defines a prototypical instance of type primitiveCircle. The gray form 392-primitiveCircle is a copy that has been modified to describe a different instance. The circle in cell newCircle is defined by the other cells, which specify its attributes. To refer to the circle elsewhere in the program, a formula can reference 392-primitiveCircle:newCircle. The programmer cannot view the formula (primitive implementation) of cell newCircle, but can view and specify the other cells' formulas by clicking on their formula tabs (▤). Radio buttons and popup menus (e.g., cell lineForeColor) provide a way to reliably enter constant formulas when only a limited set of constants are valid.

VADT forms define not only types, but also the data and behavior of objects. User-defined objects (and types) are constructed by the programmer on VADT forms and then used in the same way as built-in objects like the circle. To implement a new type of object, the Forms/3 programmer provides cells and formulas to construct one (or more) prototypical objects which, as one would expect on a spreadsheet, respond with immediate visual feedback as each new formula is entered. The formulas specify the internal data of the object, how it should appear visually on the screen, and operations that it provides.

In addition to cells, Forms/3 provides collections of cells as matrices (groups of cells in an indexable grid) and abstraction boxes (unordered groups of cells and matrices). Collectively, cells, matrices and abstraction boxes are called referenceable objects (ROs) because they are the programming objects that can be referenced in a formula. Because the term "referenceable object" is not particularly evocative, we will sometimes refer instead concretely to the possible ROs for greater clarity, e.g., "cell or cell group."

The internal data of an object is defined by cells and matrices that are placed inside abstraction boxes. The default formula for an abstraction box is the composition of its components. For example, Figure 1.2 shows a stack implemented by a one-dimensional matrix (inside abstraction box Stack), in which the programmer has added the sample value "hi." Because Stack has a sample value, as soon as the formula for cell top is entered, cell top displays Stack's top element ("hi"). Likewise, the programmer specifies the stack's image to be the top element displayed above a stack of lines, and then sees how the image appears for the sample stack immediately after entering the formula. The other formulas are also programmed in this concrete way; they reference abstraction box Stack (or its contents) and immediately display their own results based on the sample. The sample values on copies of form Stack can be replaced by references

to other cells in the program, which provides the functionality of incoming parameters in traditional languages.

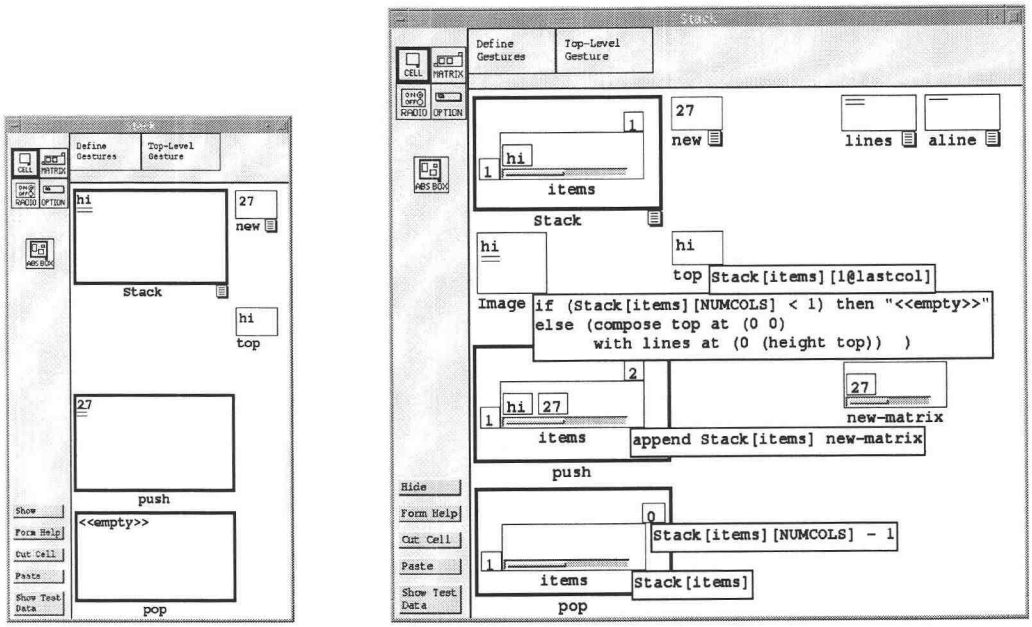

Figure 1.2 Type definition form for a stack object. (Left): The user's view of Stack hides the internal implementation and displays stacks using the formula the programmer has provided for the distinguished cell Image. (Right): The stack implementor's view of object Stack with most of the cell formulas visible. (Matrices in Forms/3 are not required to be homogeneous.)

1.3 Organization

The next chapter examines related work, and then we begin in Chapter 3 with an analysis of opportunities for explicit support for reuse already present in declarative, responsive VPLs. Chapter 4 introduces similarity inheritance, which further enables reuse of code. Chapter 5 defines a new static type system supporting similarity inheritance and providing polymorphism for user-defined operations. Chapter 6 discusses future work and Chapter 7 concludes.

Chapter 2: Related Work

This chapter discusses related work for each of the topics of the next three chapters. First, we examine previous research on reuse in general for VPL code. Second, we examine related aspects of inheritance research: applying inheritance to the spreadsheet paradigm, fine-grained approaches to inheritance, and approaches to inheritance without classes. Third, we discuss previous research in the areas of type inference and type systems for visual programming languages.

2.1 Reuse of VPL Code

We have been able to locate little previous literature on any aspect of reuse for VPLs. The previous approaches that are reported differ from ours in both the nature of the repository and the type of reuse supported. For example, reuse is facilitated in HOLON/VP [Koike et al. 1996] by allowing generic code modules to be composed by sharing objects. The emphasis is on supporting the composition of objects within a given program, however, not a general code repository. Pree's work on design books [Pree 1995] discusses reuse in the context of understanding and using frameworks. This approach has a producer-intensive focus on reuse for specific domains, and is not geared toward general support for arbitrary components.

A few VPLs provide some environmental support for a subset of the reuse questions we will identify in Chapter 3. One example is the "what kinds of components does the repository contain?" question. Fabrik [Ingalls et al. 1988] provides overview support in a simple parts bin that allows drag-and-drop access to components filed under different categories. While this is an effective strategy for easily categorized components, it does rely on the component producer to provide meaningful categorizations. National Instruments™ LabVIEW® provides similar support by placing component icons in a

hierarchical menu. The producer, to add a component, must define connectors (identifying the inputs and outputs) and save the component into a library (which determines its categorization). The producer is also encouraged to give the component a custom icon. Pictorius® Prograph CPX provides a hypertext class hierarchy browser that automatically includes new classes as they are created. The browser requires no extra producer work, but includes code from only the current project.

The Jacquard editor [Gorlick and Quilici 1994] addresses the question of finding relevant components. Each user action may narrow the search space by constraining the input or output of an empty socket (component placeholder). The user can then zoom in on an empty socket to examine the compatible components. If the number of possible components is still large, they are displayed hierarchically. To use this technique effectively, the consumer must be certain of the number and types of inputs and outputs for the needed component.

Vista [Schiffer and Fröhlich 1995] helps the consumer figure out how to use components by making "public processors" explicit. Public processors are the (sub)components that the producer anticipates the consumer will modify via substitution. Public processors therefore specify expected reuse of the component.

In the field of software reuse outside of VPLs, recent research has emphasized systematic reuse — incorporating reuse into the business of software development — with important responsibilities expected of the producers, consumers, and management [Mili et al. 1995; IEEE Software 1994; Sitaraman 1996]. While these roles are appropriate for software professionals, the context of our approach is to enable reuse in situations where institutionalized support is not available, such as for informal or end-user repositories.

2.2 Inheritance

Perhaps the most widely-used device for reuse at the level of code sharing is object-oriented inheritance. The concept of similarity inheritance is related to object-oriented inheritance, although we do not claim that our approach is object-oriented because it does not fall under standard definitions of object-oriented programming (for example, that of [Wegner 1987]). Specifically, we do not require dynamic binding of message to method and our approach is declarative, so there is no notion of mutable instance variables.

2.2.1 Combining Spreadsheets with Object-Oriented Programming

There has been little work to date on approaches to inheritance for spreadsheets and more advanced spreadsheet languages, perhaps because there has been only a little work that incorporates support for objects. Commercial spreadsheets provide support only for a few built-in types—numbers, Booleans, and strings—as first-class values, and do not provide a formula-based mechanism allowing users to add new types of objects. Although some spreadsheets gain partial support for additional objects through the use of macro languages and incorporation of other programming languages (such as Visual Basic), these approaches do not maintain a seamless integration with the spreadsheet paradigm, because they use notions such as global variables, state modification, and imperative commands in a language different from the formula language of the spreadsheet.

A few research spreadsheet languages have also incorporated external languages to support object-oriented features. ASP (Analytic Spreadsheet Package) is a spreadsheet language in which every cell can be any object, and every formula is written in Smalltalk code [Piersol 1986]. Smedley, Cox, and Byrne have incorporated the visual programming language Prograph and user interface objects into a conventional

spreadsheet for GUI programming [Smedley et al. 1996]. Both of these approaches add some of the power of object-oriented programming, but do not enforce consistency with the value rule, since global variables and state-modifying mechanisms circumvent it.

C32 [Myers 1991] is a spreadsheet language that is part of the Garnet and Amulet user interface development environments [Myers et al. 1990; Myers et al. 1997]. C32 uses graphical techniques along with inference to specify constraints in user interfaces. C32 does not itself feature the graphical creation and manipulation of objects. Instead, this function is performed by another part of the Garnet/Amulet package. The combination of tools in the Garnet/Amulet package features strong support for programming with built-in GUI objects via visual techniques, but does not support any other kinds of objects, which must be written and manipulated in Lisp/C++.

Some research spreadsheet languages have moved toward expanding the types of objects supported without the use of external programming languages. One of the pioneering systems in this direction was NoPumpG [Lewis 1990] and its successor NoPumpII [Wilde and Lewis 1990], two spreadsheet languages designed to support interactive graphics. These languages include some built-in graphical types that may be instantiated using cells and formulas, and support limited (built-in) manipulations for these objects, but do not support complex or user-defined objects.

Penguims [Hudson 1994] is a spreadsheet language for specifying user interfaces. Penguims supports composition of objects by collecting cells together, and formula inheritance at the object level. Unlike our work, it employs several techniques that do not conform to the spreadsheet value rule, such as interactor objects that can modify the formulas of other cells, and imperative code similar to macros.

2.2.2 Fine-Grained Inheritance

Although most approaches to inheritance operate at the granularity of entire classes or objects, there is some research exploring inheritance at finer granularities. Mixins [Bracha and Cook 1990] are a technique for providing inheritance on a more fine-grained scale than whole classes. Also known as abstract subclasses, mixins are partial classes that exist only to be inherited by other, complete classes. They usually define just a small piece of functionality and, combined with multiple inheritance, can cut down on the code duplication that arises when the language allows inheritance only at the class level. Another approach to fine-grained inheritance is found in the language I^+ [Ng and Luk 1995]. I^+ inheritance is not determined by subclassing, but by explicitly listing the methods to inherit.

The fine-grained aspects of similarity inheritance are closer to the I^+ approach than to the mixin approach. Some differences however, are that similarity inheritance allows even finer-grained inheritance than methods, and is flexible enough to allow mutual inheritance. Also, our approach is particularly focused on maintaining attributes important to spreadsheet languages, such as concreteness and immediate visual feedback, attributes that are not present in I^+.

2.2.3 Inheritance Without Classes

Similarity inheritance supports inheritance without subclassing or subtyping, because we believe these concepts introduce an unnecessary cognitive burden on the programmer or end user. Without subclassing or subtyping, the programmer need not spend time contemplating the best relationship among classes, organizing subclasses into a hierarchy, or understanding the subtle distinctions among subclasses, subtypes, and interfaces. We believe these concepts are beyond the capabilities of end users with no

formal programming training, and even for programmers, they require valuable time and effort.

Traditionally, object-oriented inheritance is a sharing mechanism between a class and its subclass. In contrast, prototype-based languages let programmers work directly on the objects themselves. Our approach extends the prototype idea of concreteness to the spreadsheet paradigm, by allowing programmers to see and interact with objects as directly as a spreadsheet user interacts with numbers and text in a typical spreadsheet, while also including the power of inheritance.

In most prototype-based languages, inheritance is accomplished through *delegation.* With delegation, if an object cannot handle a message directly, it delegates it to its parent object, which in turn handles it or delegates it to *its* parent, and so on. Prototypes remove the need for the concepts of class, subclass and instance since any object can be used as the basis for defining a new object.

Self [Ungar and Smith 1987; Ungar et al. 1991a; Ungar et al. 1991b] is a prototype-based language that uses a parent slot in each object to keep track of which object it inherits from. Multiple inheritance is provided by multiple parent slots that may be given priority levels. Name conflicts are resolved first by priority level, then by a special tie-breaker rule and if conflicts still occur, a run-time ambiguous message error is generated. Parent slots can even be altered dynamically, a technique the Self developers call dynamic inheritance. Because prototype languages are known for their lack of structure compared to class-based languages, Self uses traits objects as an optional technique to factor out common behavior. Similar to mixins, these objects contain only the shared behavior and data to be inherited by other objects.

ObjectWorld [Penz 1991; Penz and Wollinger 1993] is a language that, like Forms/3, uses visual mechanisms to emphasize concreteness and does not use delegation. However, unlike our approach, ObjectWorld does not use any inheritance

mechanism, instead achieving code reuse through object composition combined with automatic message propagation. An important difference between our similarity inheritance approach and most prototype-based languages (including Self and ObjectWorld) is that our model does not use any sort of message passing.

Kevo [Taivalsaari 1993] is one of the few prototype-based languages that does not use message passing. Kevo emphasizes the concreteness and self-sufficiency of objects. Operations can be marked as applying to individual objects or to *clone families* which are groups of similar objects automatically inferred by the system. Thus Kevo does not require a designated parent prototype for a collection of objects, but there are no change propagation mechanisms for objects outside the clone family. Kevo approximates multiple inheritance and fine-grained inheritance via a cut/copy/paste metaphor, but changes to the original code do not propagate, and must be recopied and pasted by the programmer.

2.3 Type Inference

Because similarity inheritance allows the programmer a great deal of flexibility in the process of creating program objects, it relies on extensive language support to help the programmer manage programs. Static type inference is one of the techniques we provide to support the programmer. Existing approaches to types are not entirely compatible with similarity inheritance, however, so we developed a new approach that is fully compatible. This section reviews some of the issues involved in static type inference in general and then describes the current state of static type inference in the area of VPL research.

2.3.1 Type inference in textual programming languages

Languages differ in the degree of type checking they perform and when it is performed. *Dynamic typing* refers to type checking performed at run-time. *Static typing* refers to type checking performed at compile-time. Static typing's advantage is that it detects type-related programming errors earlier, but it is also considered less flexible. More flexibility can be added to a statically typed language through *polymorphism*, which allows code and data structures to work with more than one type. While polymorphism is inherent in object-oriented languages, it is also possible in other language paradigms.

A statically typed language may have explicit type declarations or it may rely on type inference to reconstruct the types of variables and values in a program, relieving the programmer of the burden of declaring these types. Because of the advantages of static typing, much work has been accomplished in extending type inference algorithms originally written for functional languages [Milner 1978] to other language paradigms. As type systems become more sophisticated, it also becomes more challenging to communicate effectively with the programmer about type errors since not every programmer will be an expert in type theory.

2.3.1.1 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 the coarse-grained level of classes or prototypes.

Imposing a static view of types on a language with inheritance sometimes leads to problematic theoretical issues. These issues arise because 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 substitutivity of subtypes for supertypes but does not allow overriding in order to specialize a subtype. [Castagna 1995] has argued that such overriding is a useful mechanism and should not be ruled out of a language. The problem with having both is 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 *super*types (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 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 1997]. 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]; restricting a language to *invariant* specialization, in which a subclass method's type signature must match that of the parent class method, as in C++ [Stroustrup 1992]; incorporating templates or generic types; 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.

F-bounded polymorphism [Canning et al. 1989] is used to model the inheritance that can happen between two classes of objects that are not in a subtype relationship. It allows recursive type definitions and supports polymorphism over all objects having a specified set of methods. This model expands on the bounded polymorphism model [Cardelli and Wegner 1985], which did not allow inheritance apart from subtyping. Bruce's *bounded matching* [Bruce et al. 1995; Bruce et al. 1997] implements the equivalent of F-bounded polymorphism. To use bounded matching, the programmer provides a type that all suitable types must match in order to type-check correctly. The requirement for a match is that a type supports at least the methods of the type it matches. For example, an insert method for a binary search tree class might take as argument an object whose type matches Comparable, where Comparable is defined to have equal, greater-than and less-than methods that operate on the same type as their receiver. Formally, two object types, ObjectType$\{m_1:\tau_1; ...; m_k:\tau_k\}$ and ObjectType$\{m_1:\tau_1; ...; m_n:\tau_n\}$, are said to *match* if and only if $n \leq k$, where m_i is a method with type signature τ_i. The type expression MyType is used as the type of the *self* variable. The matching technique succeeds in separating subtypes from subclasses but also promotes the use of abstract types that define only the minimum required for matching.

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, and while Haskell reasons at the granularity of interfaces (groups of operations), it does so at the added cost of type class declarations.

2.3.1.2 Understandability of type inference results

The intent of static type inference is to ease the programmer's burden by both eliminating type declarations and preventing type errors from reaching run-time. Unfortunately, figuring out what is wrong when a program does not type check correctly can be a difficult task for a programmer.

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 2.1 and Figure 2.2. 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 even 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 2.1 A small program [Schwartzbach 1997].

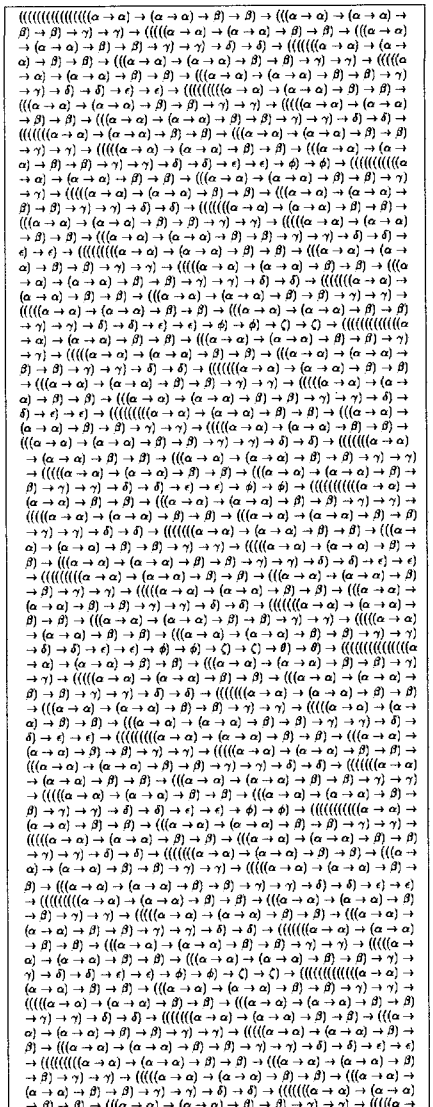

Figure 2.2 Type of the program in Figure 2.1 [Schwartzbach 1997].

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 [Duggan and Bent 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 1998]. 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 representing argument and result types (see Figure 2.3). Polymorphic types are represented by multi-striped blocks with each stripe representing a different type. A scalability issue, however, 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.

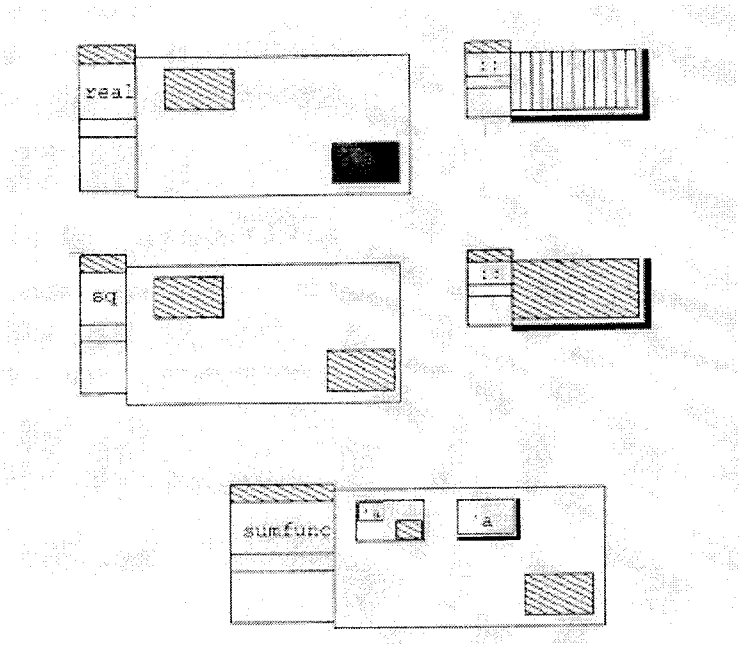

Figure 2.3 Type visualization from [Jun and Michaelson 1998]. In the paper describing the approach, colors were replaced by these black and white line patterns. (The label "real" is not a type, but rather a name for the function.) In this example, the vertical striped block represents a real type, and a diagonally striped block represents an int.

2.3.2 Type systems in VPLs

Few VPLs use explicit type declarations. The reason may be that many VPLs have as a goal reducing the number of mechanisms required to specify a program, and requiring explicit declarations seems at odds with this goal. 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, dynamic type checking can indeed produce immediate feedback about type errors in many cases, because at level 3 liveness and above, "run-time," "translation time," and "program-entry time" are intertwined. For example, in spreadsheet languages, 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.

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, and hence soundness and completeness are preserved. 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 ceases to produce output. This 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.

The type system of the functional VPL VisaVis [Poswig et al. 1992; Poswig and Moraga 1993] differs from the above in that VisaVis incorporates *implicit less ad hoc polymorphism*. *Ad hoc polymorphism* means that, for each different monomorphic type a function supports, a different implementation is required. That is, only a one-to-one relationship is present between a function's supported types and its implementations. For example, overloaded built-in operators such as "+" use ad hoc polymorphism. (Explicit) *less ad hoc polymorphism* allows a many-to-one relationship, in which a single implementation can be shared by a set of several (explicitly-declared) types. Implicit less ad hoc polymorphism infers the elements of each set. This approach is similar to the implicit aspects of type classes of Haskell. Some differences are that in VisaVis no inheritance-like structure is supported, no explicit declarations are required, and new user-defined type classes cannot be added.

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, but places some restrictions on subtypes such as invariant method signatures (subclass method signatures must exactly match the type signatures of the class methods) and requires explicit declarations of upper bounds on the types of method arguments and results.

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; Ludolph 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, which 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 (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, type information does not seem to propagate beyond the pins that are directly connected.

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; Mishra 1998]. 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 the

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.

A common limitation in many of the VPLs' type systems discussed above 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.

Chapter 3: Analysis of Opportunities for Explicit Reuse Support in VPLs

The main motivation for similarity inheritance is code reuse, in the form of code sharing, but other kinds of reuse are also useful. This chapter explores the kinds of reuse opportunities that are present, even without inheritance mechanisms in the language, for supporting evolving, informal repositories of VPL code.

3.1 Component Reuse

It has long been clear that if programmers would reuse code more often, rather than reimplementing the same logic again and again, there would be great potential for software development cost savings, including not only the cost of creating the code, but also the costs of documentation, debugging, testing, and maintenance. Unfortunately, enabling programmers to reuse code effectively has not been easy. There are many reasons why this is true, including the lack of tools for finding and composing reusable code and programmers' perceptions that writing code is easier than locating it, determining what it does, and finding out if it works [Tracz 1995].

Even though the problems with reuse have been significant, there have still been some assumptions upon which potential solutions could be built. These assumptions have included the notion that a repository of code was owned by an entity that could devise standards and procedures for including code in the repository; the notion that code producers would be required to conform to these standards and procedures before their code would be allowed into the repository; and the notion of a structured, mature repository, organized, for example, into a hierarchy of categories. However, the recent advent of informal, evolving, shared repositories of code for collaborative development

violates all of these assumptions. The use of the Web as a repository of shared code is a prime example.

The notion of informal, evolving repositories seems to be of rising interest for visual programming languages (VPLs). For example, one of Visual AgenTalk's goals is that "Environments should allow [end-user programmers] to share, with very little effort, programs and program fragments" [Repenning and Ambach 1996]. For VPLs with this kind of goal, we have been working on an approach to facilitate code reuse under such an informal, evolving repository, and in this chapter we present our results.

We will confine our definition of reuse in this chapter to the use of an existing code component in place of creating a new component. In general, a *component* is any artifact of the software process; we will concentrate on code components, but our approach is not limited to code; it also can incorporate on-line documentation, test plans, specifications, and other artifacts. We use the term *repository* to mean a collection of components. To distinguish between reuse tasks, even if they are performed by the same person, we use the term *producer* to mean the programmer who is building reusable components, and the term *consumer* to mean the programmer who is interested in using these components. Finally, *packaging* tasks are the work traditionally required of producers to prepare a component for inclusion in a formally-controlled code repository, such as conforming to standards, preparing documentation, providing test suites, etc.

In this chapter, we will assume an evolving repository that consists of components written using a VPL. We imagine several possible scenarios such as Web-based repositories of end-user-produced shareware, evolving repositories of code written by novices, or informal repositories for new software projects early in their development. In these cases, it is reasonable to assume a repository numbering in the hundreds of components. Because of our interest in informal, evolving repositories,

these are the only two assumptions we make. In particular, we emphasize that our approach is designed to work even in the absence of:

- A managing body to enforce standards and oversee the repository.
- A set repository structure such as a hierarchy of component categories or collections.
- Component producers who meticulously provide component packaging to aid the consumer.

While each of these provides valuable assistance to the consumer, informal, evolving repositories do not usually feature this level of support.

3.2 Concrete manifestations of four fundamental reuse problems

[Biggerstaff and Richter 1989] identify four fundamental reuse problems a successful reusability system must address:

1. Finding components
2. Understanding components
3. Modifying components
4. Composing components

Given no requirement for classification or packaging by the code producer and a constantly evolving repository, how can a consumer deal with these problems? To focus our investigation into this question, we have devised several concrete versions of each of these problems from the consumer's perspective, in the form of an initial assumption (such as "I am not familiar with this repository") and a goal (such as "what kinds of components does this repository contain?"). This chapter shows how our approach helps the consumer with some of these goals.

3.2.1 Finding components

- I know precisely what I want; is it here?
- I know generally what I want; which components are relevant?
- I am not familiar with this repository; what kinds of components does it contain?

- There is a component that I've used before from this repository, but I forget the name and several other details of it. Where is it?
- This component is not quite what I need; which other components are similar?

3.2.2 Understanding components

For this set of questions, the initial assumption is that the consumer has somehow narrowed down the search to some potentially useful components for the task at hand.

- How do I choose from among these similar components?
- What does this component do (in the most general sense)?
- Does this component do what I need?
- How does this component do what it does?
- Do I need to modify this component?
- What will I save by using this component?

3.2.3 Modifying components

At this point, the consumer is considering one potential component, and is deciding whether and how to customize it.

- Can I change this component to suit my needs?
- How do I make the changes?

3.2.4 Composing components

The consumer is considering one or a group of potential components. These questions relate to how to fit them together with each other and with the application in which the consumer is trying to place them.

- How do I use this particular component?
- Are these components compatible?
- Which version of this component should I use?

The work presented in this chapter concentrates on eight of the above questions.

3.3 Answering reuse questions

Our strategy in answering reuse questions for evolving repositories of VPL code is to take advantage of the integration between the language and environment in ways that remove the assumption that producers must package their code for reuse, but also give the consumer help that would not be available in a traditional textual environment. To experiment with our ideas, we have integrated a prototype into the Forms/3 language environment. The basic reusable component of Forms/3 programs is the form. Saved forms have previously only been accessible by file name via a file dialog box. We have incorporated interactive access to a code repository in Forms/3, including capabilities aimed at answering some of the consumer questions we identified in Section 3.2.

3.3.1 Overview

I am not familiar with the repository; what does it contain?

In our prototype, the consumer encountering the repository for the first time can call up an interactive overview of it by pushing the "Repository" button on the main Forms/3 window. A repository window opens (Figure 3.1), in which a large scrolling pane displays a component graph. Each node in the graph represents one component and edges indicate some sort of relationship between components. Edges can indicate dataflow (which includes calling and shared-data relationships) or similarity relationships which will be described in the next chapter. (In the current prototype, similarity edges have not yet been implemented.)

Every form that has been saved in the Forms/3 directory hierarchy with world-readable permissions is included in this component graph—no special tasks are required to prepare the code for submission to the repository. The presence of an edge is determined from derived information, made possible by the integration of the language with the language environment. This derivation not only relieves the producer from

having to document the component's relationship to other components, but also guarantees that the relationship information will not become obsolete as changes are made.

Nodes in the graph are labeled with the name of the component. More details about the origin of individual components are easily accessible in a popup window. In Figure 3.2 the consumer is looking at the detailed view of a node, and can see the component's location, author, date, and any packages to which it belongs.

The Forms/3 language uses packages (similar to Ada packages) to group related forms into larger units. The consumer can browse the list of packages found in the repository as another way to become familiar with the repository at a coarser granularity. The list of these packages can be pulled down from the query menu bar (see Figure 3.1, top left).

Recall our assumption of a repository with hundreds of components. To support a larger repository, more sophisticated visualization and browsing techniques would be required, for example [Bederson et al. 1996b; Chalmers et al. 1996; Johnson and Schneiderman 1991; Spenke et al. 1996]. However, these systems are typically oriented toward textual data. VPL-specific features such as browsing a VPL's source code according to its visual characteristics might be able to draw from the work on visual databases [Corridoni 1996; Egenhofer 1996], but we have been unable to find reports of any research to test this idea.

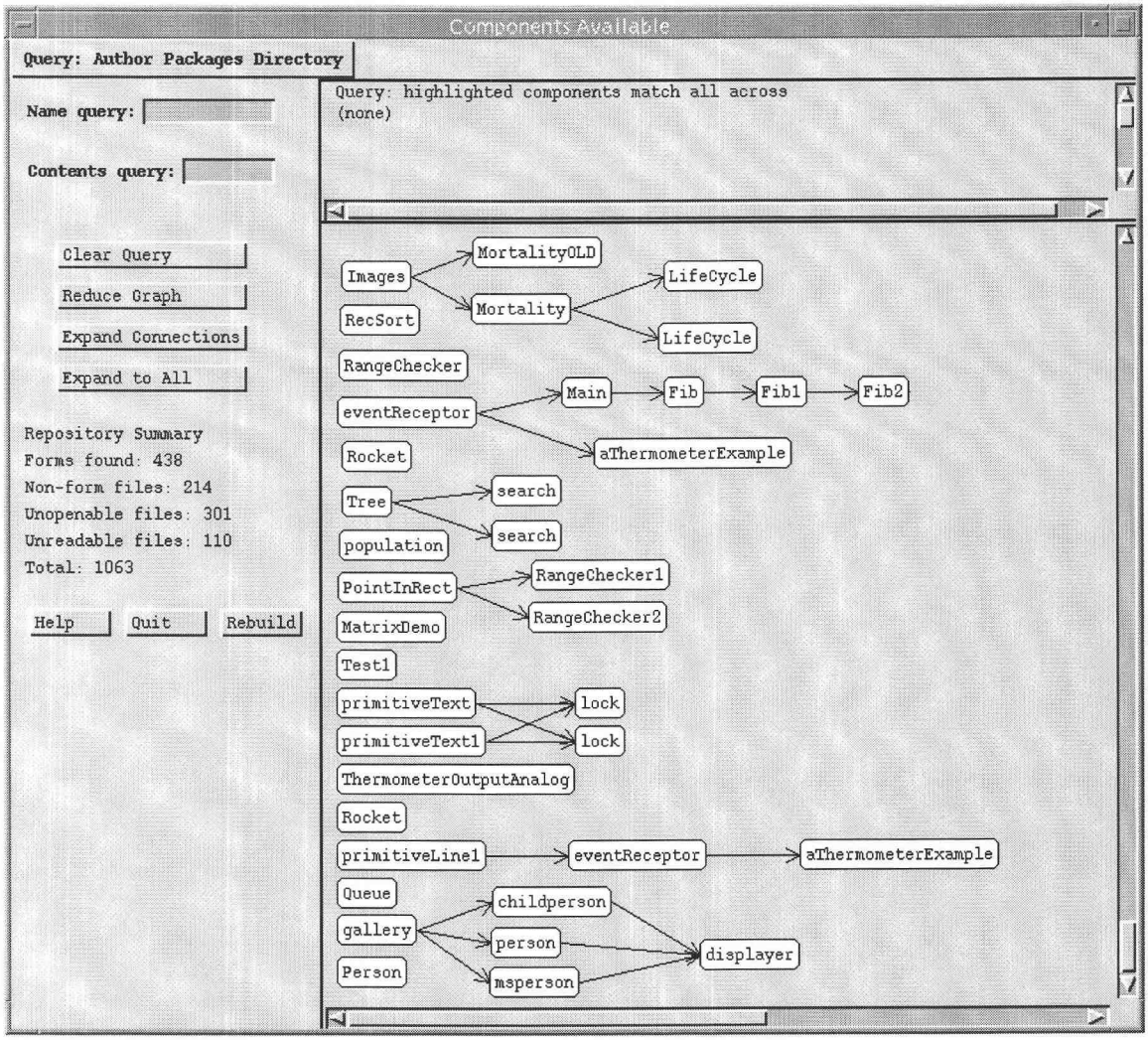

Figure 3.1 Repository overview. The large scrolling pane displays components as nodes connected by dataflow edges.

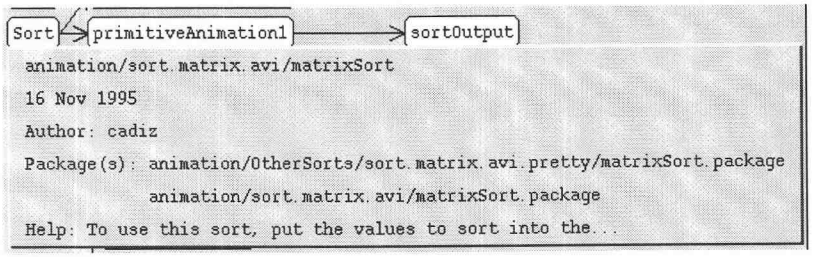

Figure 3.2 Detail view of a repository component. This detail view is pulled down from the Sort component (upper left), and shows file name, date, author, packages, and the first line of the help file.

In order to support the capabilities described thus far, an environment either needs full parsing capabilities or a source-level understanding of the stored executable code. This understanding is necessary to automatically derive relationships, thus freeing the producer from having to explicitly document them, a task that would otherwise be part of the packaging process. Adding parsing capabilities to a programming environment is not difficult for most textual languages, if there is an accessible grammar specification. For visual languages, visible grammars and parsing methods are still an area of active research. The strategy used in the Forms/3 prototype is to operate on the executable code, which is stored in a tokenized representation.

3.3.2 Search

I know generally what I want; which components are relevant?

Suppose the consumer needs to sort some products according to price and is looking for a component that performs sorting. The interactive query facility currently allows queries such as on name substring or any component text. For example, the consumer can type "sort" into the name field to highlight all nodes that include the substring "sort" and then press the "Reduce Graph" button to hide all nodes that are not

selected. The query facility also allows searching on author, package and directory, and all can be combined. Future work on our approach could allow consumers to search based on other attributes such as the types (inferred according to the axioms presented in Chapter 5) of a component's inputs and outputs.

I want a component that I've used before, but I forget the name and other details. Where is it?

Suppose instead that the consumer has a particular sort in mind. There are many components with "sort" in the name in the Forms/3 repository; upon seeing this, the consumer may continue to refine the query on this subset of the repository using other criteria. For example, the consumer may remember working on the component with two other programmers, and thinks one of them may have been the producer of the sort. Adding both producers to the query (called authors in the query environment) by selecting them from the query menu turns off the highlighting on nodes not associated with those authors. The final results of this query are shown in Figure 3.3.

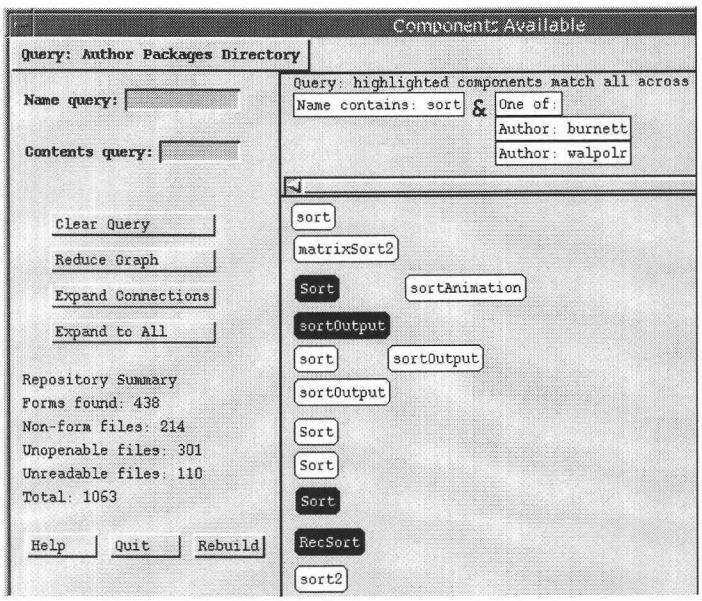

Figure 3.3 Query results. Before the authors were added to the query, all of the components here appeared highlighted.

I know precisely what I want; is it here?

Finally, suppose the consumer is specifically looking for a bubblesort. Queries can also include searches over the textual contents (comments, cell names, formulas, and even sample values) of a component and any associated on-line documents such as help files, test plans, and specifications. The consumer can type "bubblesort" into the "Contents query" text box to highlight any component that contains this term in its textual contents or associated documents.

Although the query facility described in this section is not unusual in the features it offers consumers, it is innovative in two ways. First the approach supports the ability to query without requiring any packaging on the producer's part; and second, its tight integration into the language environment prevents the consumer from switching between separate tools for searching for code, testing it, and changing it, as will be seen in the next section.

3.3.3 Exploration

What does this component do?

When the consumer is ready to investigate the purpose of a particular component, a repository browser for a traditional textual language (e.g., CodeFinder [Henninger 1994]) is likely to display the component's source code, but to provide no additional support for exploring the component. This is where three features found in many VPLs can be leveraged to provide support for component exploration, allowing the consumer not only to view the source code, but also to interact with executions of it.

The first two of these three features are sample values and immediate feedback. Returning to the sorting example, to explore a component of interest in the Forms/3 repository, the consumer double-clicks on its node in the graph. Without switching environments, the consumer immediately sees not only the code (formulas), but also its

effects on sample values. For example, see Figure 3.4. (The formula tab hanging from the bottom of the "input" matrix indicates that new input can be entered here to sort values that are different from the sample.)

These sample values give concrete examples of the computations, illustrating the component's function. Sample values are provided by the producer of the component, but since providing sample values is the natural way to program and test in Forms/3, no special step or extra work is required of the producer. This leveraging of sample values to help answer the "What does this component do?" question can be accomplished by any VPL, such as by-demonstration and spreadsheet-based languages, that supports sample values.

Does this component do what I need?

The information provided by the sample values is greatly enhanced by the presence of immediate semantic feedback (found in responsive VPLs) because the consumer can interact with the system by entering new sample values and immediately see the results. Different sample values suited to the consumer's needs, such as an input matrix that includes duplicate values, may give the consumer more confidence in the suitability of this component.

How does this component do what it does?

This question arises in customization situations. When the consumer displays hidden cells (local variables) and detailed comments and then allows the system to move through time as in Figure 3.4, the synchronized changing values of the cells resulting from responsiveness automatically function as a low-level visualization of the code. In fact, this same interaction style even allows the consumer to change any formula desired and immediately see the results, thereby allowing experimentation with changes and the development of customization ideas. VPLs with somewhat less liveness can still offer a

degree of interaction, even though more steps (such as pressing an execute button or asking for individual values) will be necessary.

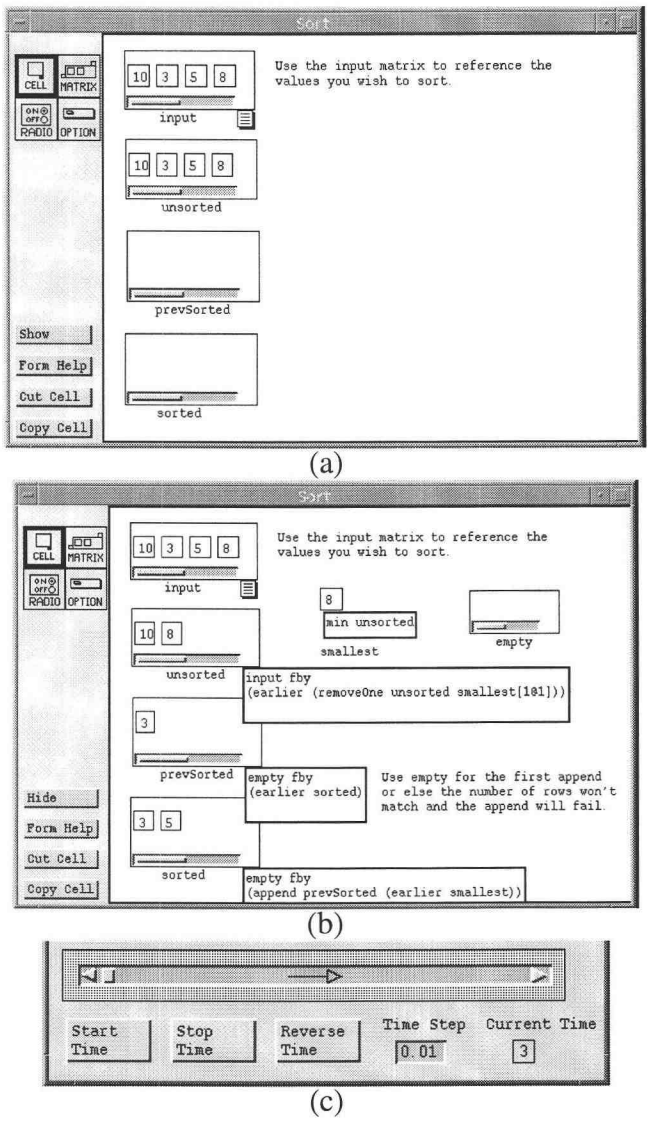

Figure 3.4 Sort component. (a) shows the sort component as it appears when first loaded. In (b) the consumer has displayed most of the formulas and all the hidden code and comments. By manipulating the timeslider (c), the consumer can watch the sort progress.

While no special effort is assumed of the producer in the Forms/3 repository, if any extra work is associated with a component as it matures, the results of that work are available in the same environment. One such extra is algorithm animation, the third feature of special utility in VPLs for answering the question of what a component does. Algorithm animation is "the process of abstracting the data, operations, and semantics of computer programs, and then creating dynamic graphical views of those abstractions....[it involves] program views that conceptually portray how a program works" [Stasko 1990]. In contrast to the detailed visualizations discussed above, an algorithm animation allows a higher-level view of what a program does.

Forms/3 includes full support for algorithm animation [Carlson et al. 1996], and whenever a producer of a component in Forms/3 has prepared an associated animation, the component graph will show an edge to an animation component. In case the consumer has reduced the graph (as in the example query above), the "Expand Connections" button adds back into the graph all components that have an edge to or from those already in the graph. Figure 3.5 shows a part of the graph after expansion. The consumer can click on the associated animation directly from the graph to see a high-level visualization of the original component. For example, Figure 3.6 shows an algorithm animation in the midst of execution. The consumer controls the animation's speed and direction while the algorithm and the animation execute synchronously. Algorithm animation is one way to cultivate abstract understanding, which is an important factor in the ability of novice programmers to reuse code [Hoadley et al. 1996].

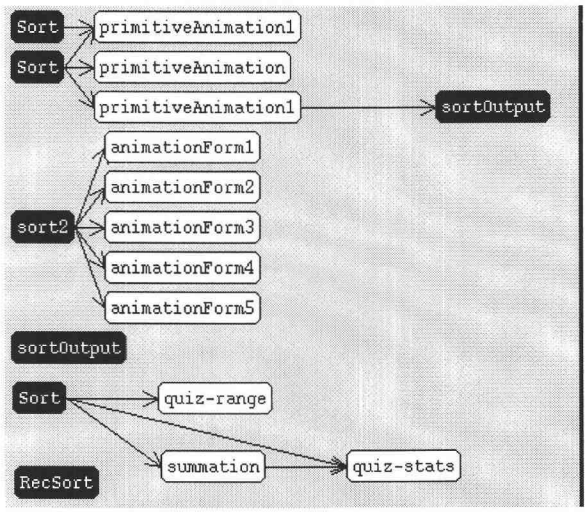

Figure 3.5 Expanded graph. A portion of the graph after it has been expanded to show connections with other components.

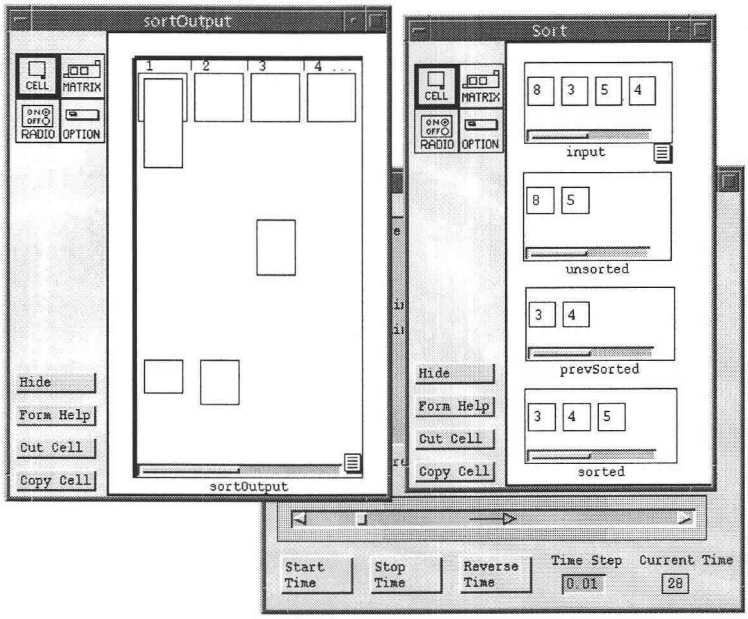

Figure 3.6 Sort animation.The consumer has loaded the animation and is controlling it with the time-slider at the lower right.

3.3.4 Use

How do I use this particular component?

A recent study of professional Smalltalk users [Rosson and Carroll 1996] observed that consumers make extensive use of previous usage contexts when figuring out how to use unfamiliar components. The dataflow graphs of the overview automatically display these contexts, and the consumer can interact with any of them using the same mechanism described earlier to find algorithm animations.

For example, in Figure 3.5 the consumer can see that one of the highlighted sorts is connected to a component named "quiz-stats." The consumer can load the sort and this context by double-clicking both nodes on the graph. Or, if the producer has set up a package, double-clicking on either node will produce a dialog box allowing the consumer to choose between loading the component alone or an entire package.

By running the quiz-stats program, which uses the sort component, the consumer can see by example how to use the sort. In Figure 3.7, the quiz-stats program has run and the consumer can see that it uses a Boolean value to indicate whether the sort has finished. From this example, the consumer concludes that the same Boolean test would be useful in new programs that use this sort.

We believe this ability to browse, retrieve and experiment with usage contexts is critical to allowing consumers to successfully reuse code that has not been packaged by its producers. This ability permits an alternative to documentation and standardization procedures that would normally be producer tasks, a feature that is especially important in immature repositories of developing code and in cooperative repositories not "owned" by any organization.

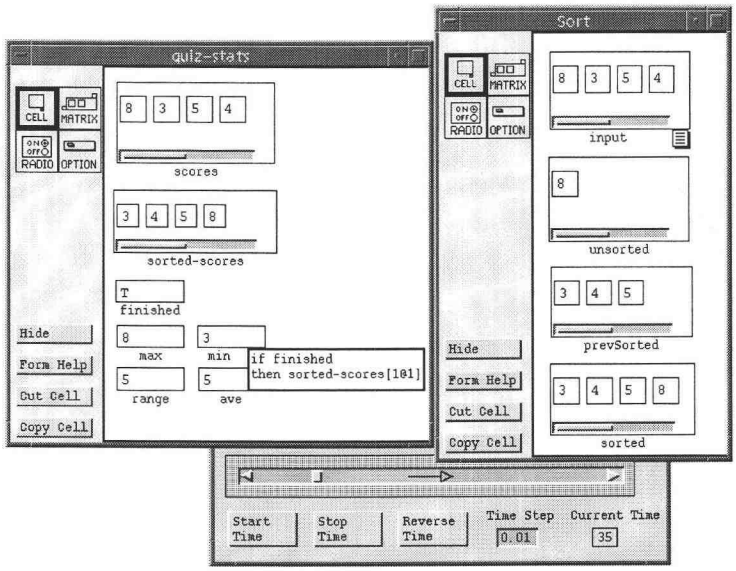

Figure 3.7 The quiz-stats program. This screenshot shows time advanced far enough so that the cell finished has a true value. The consumer can see that the statistics are computed only if the sort has finished.

3.4 Implementation Status

The user interface for the repository presented in the examples throughout this chapter was written using the Garnet user interface development environment [Myers et al. 1990]. Although functional, the graph portion of the interface turned out to be too slow for practical use. After some investigation, we decided to reimplement the graph portion using the Graphviz tools [North and Eleftherios 1994] from AT&T. Although requiring a separate process and socket communication, the new implementation resulted in major speed improvements. The functionality of the new version is largely the same, the main difference being that the overview window of Figure 3.1 is now separated into two windows, one displaying just the graph and another displaying the query interface. A few improvements were also added into the new interface, such as the ability to zoom in and out, open a reduced (birdseye) view of the entire graph, and load a package

directly from the popup menu rather than a separate dialog box. Figure 3.8 is a screen shot of the new implementation.

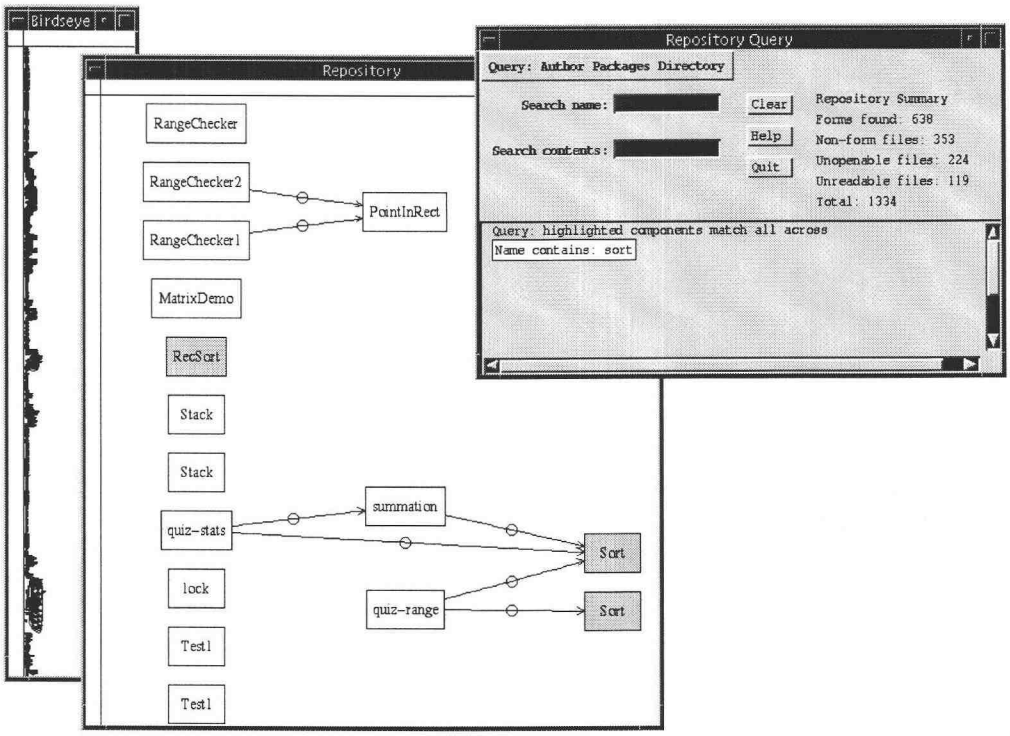

Figure 3.8 Revised implementation of the repository interface. Shown here are the birdseye view (left) repository window (middle) and query window (right).

3.5 Conclusion

In this chapter, we have presented an approach to facilitating reuse in evolving repositories of shared VPL code. An important aspect of the approach is that it takes advantage of features already present in many VPLs. The techniques presented here are suitable for locating and using components, but they do not address code reuse at smaller granularities, such as object-oriented methods. The next chapter explores the further step of integrating support for code reuse at the language level.

Chapter 4: Similarity Inheritance

In this chapter we show how the existing approach to objects in spreadsheets described in the introduction (Section 1.2) can be extended to support inheritance-like code sharing. Our motivation was to support, without adding the extra trappings of object-oriented classes, the kind of code reuse that might otherwise be achieved by copying code from one place to another. As a result, our approach emphasizes implementation over interface. Our approach, called similarity inheritance, defines "like-a" relationships between objects rather than the "is-a" relationships often associated with subclasses. Further, similarity inheritance adds code sharing mechanisms to spreadsheet languages without violating the value rule or incorporating other programming languages or macro languages. We begin by describing our new model of inheritance, independently of any language implementation. Next we present the prototype implementation of similarity inheritance in Forms/3 and describe how it implements the model.

4.1 Similarity inheritance model

In the model description, we will use object-oriented terminology to facilitate comparison with other models of inheritance, although it will later be demonstrated (Section 4.2.5) that the approach is not restricted to relationships among objects, and can be used for relationships among Excel-like spreadsheets as well.

We define the similarity inheritance model to consist of both a semantic model and a model of interaction (between the programmer and the computer). The interaction model is composed of the tuple (**C**, **F**, **L**, **R**) where **C** is the copy operation, which creates a shared definition, **F** is the formula definition operation, **L** is a liveness level 3 or higher from Tanimoto's liveness scale [Tanimoto 1990] indicating that immediate

semantic feedback is automatically provided, and **R** is a representation mechanism that explicitly includes all shared formulas and relationships in each object's definition.

Three important points about the interaction model are: (1) it separates the syntax with which the human communicates to the computer about program semantics from that used by the computer to communicate to the human about program semantics (for example allowing animations or graphical views), (2) it does not necessarily map to a static textual syntax (for example, it allows dynamic syntaxes) and (3) it depends on environmental characteristics that are not usually guaranteed by textual programming languages but are common in visual ones. Note that the elements of the interaction model are not mere editing details of an environment, but rather define the general characteristics upon which our semantics rest.

The semantic model can now be defined as follows. Each object definition D in a program is a set of behaviors $\{D_{b1}, D_{b2}, ..., D_{bn}\}$. For example, in Forms/3, an object definition is a VADT form. Each D_{bi} is a formula residing in a cell (or cell group). The symbol $\rightarrow$ (pronounced "shares with") indicates a shared behavior; the arrow points from the original version to the copied one. It defines a "like-a" relationship between the original and the copy, and describes how the programmer has arranged similarities among program objects. The semantics of $\rightarrow$ are

$$A_{bi} \rightarrow B_{bi} \Rightarrow A_{bi} = B_{bi} .$$

The operations **C** and **F** determine when $\rightarrow$ holds, as summarized in Table 4.1.

Operation	Precondition	Postcondition
C used to copy A	A is an object definition B does not exist	B exists and $\forall$ i, $A_{bi} \rightarrow B_{bi}$
C used to copy from A_{bk} to B	A and B are object definitions and A≠B	$A_{bk} \rightarrow B_{bk}$
F applied to B_{bk}	B is an object definition	$\forall$ A, $A_{bk} \not\rightarrow B_{bk}$

Table 4.1 How **C** and **F** affect "like-a" relationships. ($i \in \{1..n\}$ The table omits postconditions that duplicate preconditions.)

The first row of Table 4.1 defines large-grained similarity, and means that if **C** is applied to an object definition A, a similarity relationship[1] will be created between A and a new object definition B such that all of A's behaviors share with B (this can be abbreviated $A \rightarrow B$). Because $A_{bi} \rightarrow B_{bi}$ holds for *all* A's behaviors, additions to the definition of A will also propagate to B in order to maintain that condition. The second row defines fine-grained similarity, which allows a single behavior A_{bi} to be copied to object definition B to create a "like-a" relationship between A_{bk} and B_{bk}. **C** may also be applied to groups of behaviors, in which case the semantics are the same as if applied to each individually. The third row implies that overriding removes "upstream" sharing relationships, but not "downstream" relationships—a formula that is redefined may still share with others but nothing shares with it.

Due to element **R** of the interaction model, objects in the similarity inheritance model have the property of *self sufficiency* from the programmer's perspective, meaning that every supported operation for an object and every piece of data it contains can be determined by examining the object's own definition rather than also requiring the inspection of parent objects or descriptive classes. The implication of the **L** element of

[1][Perrone and Repenning 1998] recently proposed a technique called *analogies* that bears a surface resemblance to similarity inheritance. Their technique, however, does not create a relationship between source and copy, but is a one-time editing convenience.

the model is that the programmer, rather than working with abstract descriptions (as in the class-based model), creates and manipulates concrete descriptions, in which live objects reside. A significant difference in the concreteness just described from that of the prototype model is that in our model the sharing relationships are created statically rather than dynamically.

From this model, the granularity difference between similarity inheritance and other approaches becomes clear. Both the class-based and prototype-based models support $\not\rightarrow$ at the granularity of methods (overriding), but neither supports $\rightarrow$ at the granularity of methods, for example the ability to inherit just one method. Multiple and mutual inheritance are direct by-products of fine-grained similarity. Multiple inheritance occurs in cases such as $A \rightarrow B$, and $C_{b1} \rightarrow B_{b1}$. Mutual inheritance occurs in cases such as $B_{b2} \rightarrow C_{b2}$ and $C_{b3} \rightarrow B_{b3}$.

4.2 Similarity inheritance in Forms/3

4.2.1 Interaction model

The interaction model is instantiated in Forms/3 as follows. Operation **C** is supported by a *copy form* button, which copies the form selected in a scrolling list, and by a *paste* button on each form, which pastes selected cells onto the form. Recall that **C** is not simply an edit operation, but defines the $\rightarrow$ relationship. Operation **F** is supported by allowing the programmer to edit any formula that is visible. Liveness level **L** is level 4, so after every formula edit, immediate visual feedback is given about the edit's effect on the program. In Forms/3's representation **R**, each behavior (cell and formula), whether the result of sharing or not, is visible, which allows it to be edited by operation **F**. Shading indicates whether a form or cell is the result of sharing. Section 4.3 explains additional features of Forms/3's representation. The explicit representation of all

behaviors is a significant difference from most object-oriented languages in which inherited code is found strictly in the superclass or parent object. The yo-yo problem [Budd 1991, p. 90] is a result of this non-localization of code which requires a programmer to potentially search many objects or classes up and down the inheritance hierarchy in order to understand the behavior of a single object or class. Similarity inheritance's explicit representation prevents the yo-yo problem.

4.2.2 Large-grained inheritance

The stack in Figure 1.2 is an example of an object created from scratch. Multiple stack objects can be created with the copy operation **C**, but suppose the programmer wants not another stack but something else that is similar to a stack, for example, a queue. Taking advantage of that similarity, the programmer can start with a copy of form Stack, then modify the behavior using operation **F**. Inheritance and overriding of cell names as well as of cell formulas are allowed. A change to the operation push and some renaming of cells are all that is required to turn the copy into a Queue (see Figure 4.1).

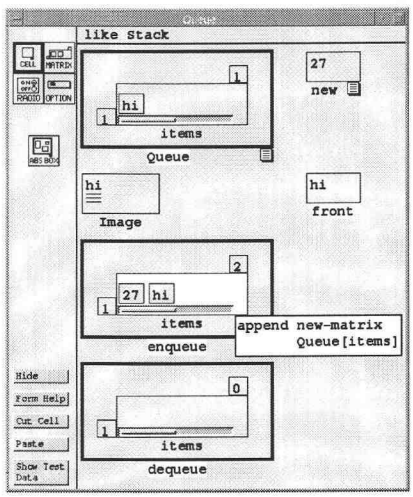

Figure 4.1 A Queue created with similarity inheritance from a Stack. Several names and one matrix are unshaded to indicate that they have been overridden. The cell Image could also be overridden to create a custom appearance for a Queue.

Note that using similarity inheritance, the programmer simply identifies that two objects are similar in implementation or purpose. In contrast to this, in a class-based language, the programmer may spend extra time wondering what the "right" relationship is between a stack and a queue. One is not a subtype of the other, yet they are similar. In fact, extra work to reorder the inheritance hierarchy may be needed in some cases just to add one new class. As we saw above, however, similarity inheritance allows the programmer to create a similarity relationship without implying "is-a" subtype or subclass relationships. Instead, it defines a "like-a" relationship. For example, a queue is *like a* stack except that new items are inserted at the opposite end.

Because of the $\rightarrow$ relationship between Stack and Queue, any change to a formula definition on Stack will propagate to Queue unless the Queue's cell formula has been overridden. For example, a fix in the formula of cell push on Stack would not have any effect on Queue, but a fix in cell pop would propagate to the dequeue operation on form Queue. Some prototype-based languages lose this ability to propagate changes to groups of objects because of their emphasis on object individuality; instead, shared parts must be abstracted out of the objects, as in Self's traits [Ungar et al. 1991a].

4.2.3 Fine-grained and multiple inheritance

As noted in the discussion of the model, the combination of large-grained and fine-grained similarity allows multiple inheritance. For example, in Forms/3 suppose a new form Deque is created via large-grained similarity from Queue. (A deque is a double-ended queue.) This new object needs to allow items to be added to either end of the queue. The programmer may notice that Stack's *push* is exactly the required behavior for Deque and can use fine-grained similarity, copying *push*, to allow Deque to inherit just that one operation from Stack. Because of the interaction element **R**, the programmer now sees the new cells as part of the definition of Deque also (Figure 4.2).

Multiple inheritance in other languages can lead to conflicts when more than one method of the same name are unintentionally inherited. By providing inheritance on the level of cells, the similarity model allows the programmer to select only the operations that are actually needed, avoiding unintentional inheritance. (If the programmer does accidentally attempt to introduce a conflict, the system provides options for resolving it at the time of the edit.)

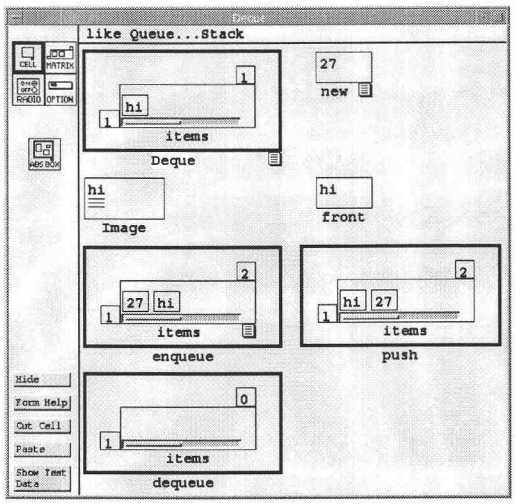

Figure 4.2 A Deque in progress. The Deque was created by copying the Queue form and the operation push from Stack. The name of the abstraction box Queue has also been changed to Deque causing the name to appear white instead of shaded.

4.2.4 Mutual inheritance

Suppose, as in Figure 4.3, someone added the new operations *size* and *empty?* to Queue. Another programmer might find those operations useful for Stack as well and copy them to the Stack form. Stack and Queue now both inherit from each other. Like multiple inheritance, mutual inheritance is not a new concept in the language, but rather a feature of the flexibility of similarity inheritance, which makes mutual inheritance

straightforward. To the best of our knowledge, similarity inheritance is the first model to support mutual inheritance.

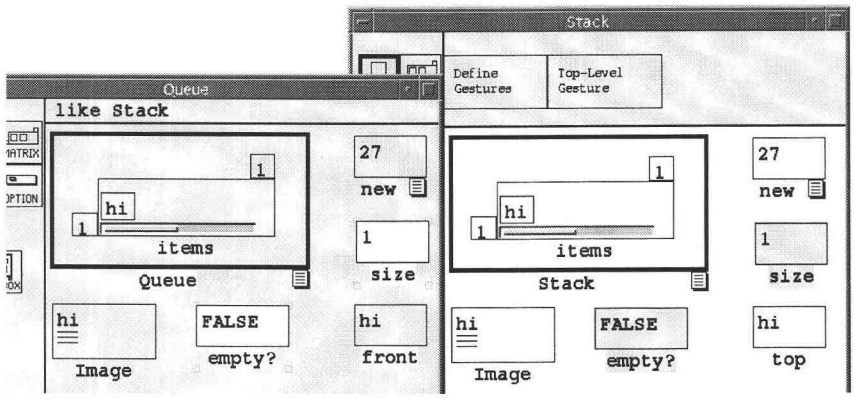

Figure 4.3 Mutual inheritance between Queue and Stack. The new cells *size* and *empty?* appear white on the Queue form where they originated, and shaded on the Stack form.

4.2.5 An end-user example

In the previous sections, we have discussed our approach from the standpoint of how it can be used to share behavior among objects. However, as has been noted earlier, the approach is general enough to allow sharing of other pieces of programs, even when there is no relationship among the types of objects involved. This allows the same approach to be used for simple formula reuse as for object inheritance, instead of prior approaches, which relied on copy/paste and "replicate" options. The advantage of using inheritance for reusing spreadsheet formulas is that the relationships among originals and copies are *maintained*, supporting automatic propagation of bug fixes and explicit depiction of relationships.

For example, consider Figure 4.4, which shows a spreadsheet (written in Forms/3) to compute course grades. Suppose the user teaches several sections of the

course, and keeps each section in a separate spreadsheet for convenience.There are two reuse situations in this example: the reuse of the formula for the top row down through the remaining rows of this section, and the reuse of these formulas in other sections. In the first case, traditional spreadsheets use a "replicate" mechanism (copy down the rows). Our system does not apply inheritance to this case; instead, like some other spreadsheets, it has a way to group cells with a common formula. It is the second case in which we apply similarity inheritance. In the second case, traditional spreadsheets use a "copy/paste" mechanism (copying into other sections), and then if the weights need to be changed, the user would have to remember to do all of the copy/pasting again. However, if a system implements copying using similarity inheritance to make the relationships explicit, as does Forms/3, then a change to the grading weights in the first section can automatically propagate to all the other students.

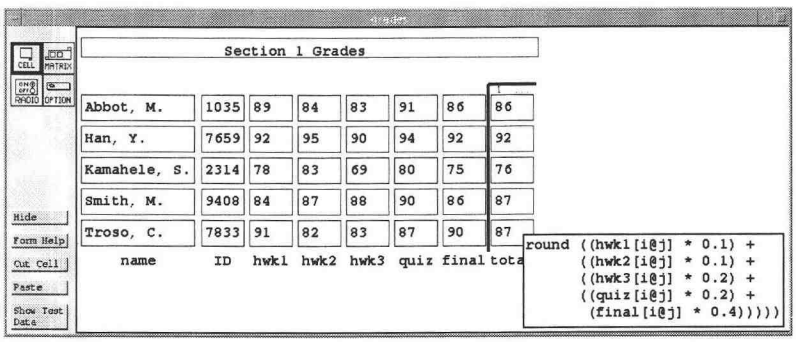

Figure 4.4 A spreadsheet to compute grades. The cells in the *total* column are grouped into a matrix and thus need only one formula (shown) to define their values. The formula computes the course grades via a weighted average.

As this example demonstrates, similarity inheritance can be used not only to maintain relationships among objects, types, and operations, but also among pieces of any sort of calculation. An attractive feature of this generality is that it affords a gradual migration path for users to move from using only simple numbers and strings in their

formulas to using more complex objects with inheritance as they gain expertise, since the same mechanism for inheritance is employed for reusing formulas in both situations.

4.3 Explicit representation

The interaction model requires the existence of an explicit representation, **R**, that explicitly includes all shared formulas and relationships in each object's definition. Previous sections have illustrated the representation of shared formulas. This section describes the representation of "like-a" relationships both between forms and between cells.

The "upstream" relationships are made explicit by legends at both the form and cell level. A legend under each cell formula lists the cell that it was directly copied from. If that cell was in turn copied from another, ellipsis follow and the name of the original cell is also given. Figure 4.5 illustrates examples of both cell and form legends.

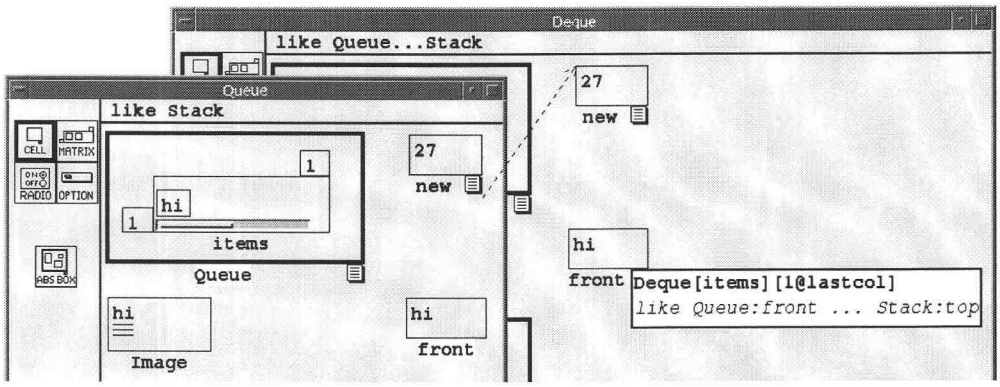

Figure 4.5 Explicit representation. The Deque form on the left has a form legend at the top indicating it is copied from Queue which in turn was copied from Stack. (If there were intermediate forms, the legend would take the form "Queue...3...Stack" and the programmer could click on the 3 for a full list.) Deque's *front* cell illustrates a formula legend. Copy dependencies among cells are also explicitly depicted with optional arrows such as the one from Queue's *new* cell.

The "downstream" relationships are represented by different kinds of arrows. At the cell level, optional similarity arrows, such as the one shown in Figure 4.5, point from a cell to the cells created from it. At the form level, additional arrows in the repository view described in the previous chapter indicate both large-grained similarity and fine-grained similarity (at least one cell) between forms (see Figure 4.6).

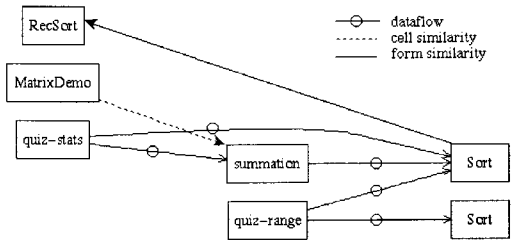

Figure 4.6 An example summary view. Only the dataflow arrows are implemented so far; the other two have been manually added to the screen shot.

To help evaluate and improve the design of **R** in Forms/3, we used a set of design benchmarks [Yang et al. 1997] that are a concrete application of several of the *cognitive dimensions* for programming systems [Green and Petre 1996]. The evaluation of this final version of the design appears in the Appendix.

4.4 Discussion

This section discusses the similarity inheritance model according to two perspectives from the literature. The first is an evaluation of the mechanism according to the concept it supports. The second is a comparison of the outcome of the mechanisms (in terms of sharing achieved) among different approaches to inheritance.

Inheritance mechanisms have been used to express a wide variety of programming concepts. [Evered et al. 1997] identify sixteen different concepts which can be (and have been) realized using inheritance (see Table 4.2). Different languages

use inheritance to support different subsets of these concepts and with different degrees of success. In order to evaluate the support for each concept, the authors list the requirements for an inheritance mechanism to provide good support for that concept. For the category of general code reuse (the concept targeted by similarity inheritance), their list of requirements is:

- selective inheritance of operation implementations
- redefinition of operation implementations with access to the inherited implementation
- multiple inheritance
- decoupling the type inheritance hierarchy from the implementation inheritance hierarchy

Modeling concepts	**Design concepts**	**Implementation concepts**
specialization	modularity	data abstraction
decremental specialization	behavior extension	subtype implementation
dynamic, incremental specialization	behavior restriction	substitutability
generalization	genericity	general code reuse
attribution	polymorphism	
components	importation	

Table 4.2 Concepts supported by inheritance mechanisms [Evered et al. 1997].

Although they provide support for many of the requirements associated with other concepts, none of the common object-oriented languages provide good support for general code reuse according to this list. Similarity inheritance aims at supporting the concept of general code reuse, so we would expect that it meet all four requirements. In fact, all of these requirements are indeed met: operations can be selectively chosen for inheritance using **C** on groups of cells; inherited implementations can be redefined using **F** and the original implementation can be also be accessed by referencing cells on the original form; multiple inheritance is supported as described in Section 3.2.3; and the

structure of inheritance relationships is not tied to a type hierarchy (as will be described in Chapter 5).

As an additional approach to comparing inheritance mechanisms, [Bardou 1996] introduces a formalism for characterizing the kinds of sharing achieved by each. In this section we informally adapt this technique as an aid to the comparison of similarity inheritance with class-based inheritance and prototype delegation.

We begin with a few definitions. A *property* is any named attribute of an object; in different languages, properties go by various names such as instance variables, methods, attributes, fields, slots or in the case of Forms/3, cells. We differentiate between two kinds of sharing[1]:

code sharing[2]: using the same code (constant value or lambda-expression/formula) for a property in two objects. If object1 shares code for property x with object2, then changing the code for object1 will change the code for object2. If the code sharing is *one-way* then changing the code for object2 will end the code sharing and will not affect the code for object1. If the sharing is *two-way,* changes to object2 also change object1.

name sharing: using the same name for the same property in two objects. Object1 shares a name with object2 if changing the name of property x in object1 causes object2's name for property x to change also.

Although the specific rules change from language to language (especially among prototype-based languages), the different sharing mechanisms can be described generally using these definitions.

Class-based inheritance: Instance variables must use name sharing. Class variables must use name and code sharing. Methods must use name and code sharing

---

[1]Bardou defines three kinds of sharing, but the third, property sharing, is not necessary for this discussion.

[2]Bardou uses the term "value sharing" to refer to the contents of a property (constant value or lambda-expression/formula). We use the term "code sharing" instead to emphasize that it is the contents (such as the formula) that is shared, not the return value or result of the formula.

among objects of the same class and must use name sharing with optional code sharing for objects of a subclass. An object must share all shareable names in a class if it shares one (we disregard names that are never inherited, such as private methods in a C++ superclass). All of these relationships are determined statically.

Delegation: Relationships are dynamically created and in some languages (such as Self [Agesen et al. 1993]) can be dynamically altered. An object must share all names in another object if it shares one. Code sharing implies name sharing, but name sharing does not imply code sharing, that is, code sharing is the default for all properties that share names, but the sharing can be broken (overridden). Code sharing is usually two-way, which can lead to the accidental corruption of a prototype instead of a clone [Meuter et al. 1996], but depending on the language can also be one-way (Act 1 [Lieberman 1986a]) or the choice can be left to the programmer (NewtonScript [Smith 1995]).

Similarity Inheritance: Relationships are created statically. The programmer can choose between sharing all properties or sharing only specific properties. Name and value sharing are independent of each other (both are shared by default, but either can be overridden).

	Relationships created	Code sharing implies name sharing	Sharing at object (class) granularity	Sharing at property granularity	Two-way code sharing
Class-based	statically	Y	Y	N	N*
Delegation	dynamically	Y	Y	N	Y**
Similarity Inheritance	statically	N	Y	Y	N

Table 4.3 Comparison of sharing achieved by various approaches.
*class variables could be considered two-way, but regular instance variables and methods are not
**in some languages

Table 4.3 summarizes the differences among these three approaches. Similarity inheritance differs from both class-based inheritance and prototype delegation by allowing name and code sharing to exist independently and by providing sharing at the granularity of properties.

Chapter 5: Type Inference

In adding inheritance to a declarative VPL, we did not wish to compromise any of the existing features of the language and particularly did not want to introduce explicit type declarations. Static type inference seemed attractive because it would allow early feedback (at edit time) to the programmer about possible errors in the program. In addition, since inheritance has traditionally been associated with subtyping, it seemed important to us to provide an approach to types compatible with similarity inheritance. Unfortunately, the previous type inference algorithm for Forms/3 [Burnett 1993; Mishra 1998] was not powerful enough for use with similarity inheritance. Neither are other approaches to type inference with inheritance for textual languages usable because of the complexity of the types. Thus we required a new approach to type inference with the following properties:

- Fine-grained inference: Most static type inference systems derive type information at the granularity of entire classes, and this level of granularity inhibits 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 communicating type errors even to professional programmers. This lack of understandability is especially unacceptable in VPLs aimed at end users.
- Power without the addition of explicit declarations: A model of type inference that is suited to both end users and programmers would allow VPLs to retain the benefits of static typing without requiring the user to engage in the programming mechanics of explicitly declaring types. In implicitly typed languages, the introduction of inheritance has typically re-introduced explicit declarations.

In this chapter, we present a model for static type inference with these properties in mind. Our model aims for understandability to both programmers and end users by basing its reasoning on concrete references to objects and operations attempted rather than

reasoning in terms of complex type names. This strategy turns out to be powerful enough to support similarity inheritance without the need for explicit declarations.

5.1 Organization of this chapter

The next section introduces a subset of Forms/3 used for formal reasoning, including the translation between full Forms/3 and this subset. Next, we briefly review the basic concepts of polymorphic type inference and introduce the kind of polymorphism we want the type inference to support. Section 5.4 presents the new model of types and the rules for determining when a program is type safe. Example derivations of type information are provided in Section 5.5 and properties and implications of the model are discussed in Section 5.6.

5.2 Core Forms/3: the subset for formal reasoning[1]

This section contains a formal definition of Core Forms/3, which supports the complete semantics of Forms/3, but eliminates syntactic sugar and other programming conveniences. Because Core Forms/3 is small, it allows the type system to be defined, without loss of generality, using a small axiom set. The main difference between full Forms/3 and Core Forms/3 are the basic formula model (cell formulas can contain only a constant, single cell reference, or composition of parts) and simplification of attributes.

[1]Section 5.2 is taken from [Djang et al. 1998] and is the work of all the authors.

5.2.1 Programming objects and notational conventions

The following define the basic programming objects in Core Forms/3:

A *program* is a set of forms.
A *form* is a tuple: (formID, modelName, ROset), where
ROset is a set of referenceable objects having unique cellIDs, and
modelName indicates the form copied from
(modelName = formID if this form is not a copy or has been renamed)
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[1], ROset, formula, value) whose ROset contains only (zero or more) virtual ROs. (defined below)
A *matrix* is a tuple: (cellID, ROset, formula) 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 the set difference ROset - {<MID>[numrows], <MID>[numcols]}.)
A *formula* is as defined in Table 5.1.

The Core Forms/3 programming objects also exist in full Forms/3, but in the full version they have additional cosmetic attributes, such as positions and borders, not present in Core Forms/3. The reason for the modelName element of forms is to explicitly track similarities among forms, which allows useful generalizations to be made during type inference. Note from these definitions that it is not possible in Core Forms/3 for a cell or cell group to exist without a formula. In full Forms/3, a cell or cell group might not yet be assigned a formula, and instead contains a special constant value "no value." Also note that matrices, unlike other ROs, have no value: rather, matrices are a flexible mechanism to support spreadsheet-like grids of cells (each of which has a value). However, we will occasionally refer to a matrix's value as an abbreviation for the set of values of the ROs in the matrix's ROset.

[1]In full Forms/3, when communicating with the programmer about type errors, each cellID is converted to its cell name if the programmer has provided one.

In addition to the above objects, type definition forms (VADT forms) support both built-in and user-defined types:

A *VADT form* is a form whose ROset includes a cell with cellID "Image", one abstraction box with cellID "MainAbs", and zero or more additional ROs. (VADT stands for *visual abstract data type*.)
An *abstraction box* is a tuple: (cellID, ROset, formula, value) whose ROset contains only cells and matrices and that 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 that is an element of another cell's ROset.
A *virtual matrix* is a matrix whose ROset contains only virtual cells and that is an element of another cell's ROset.

In the previous chapter, we have referred to a VADT form as defining objects, but recall that it also defines a (monomorphic) type. Each VADT form's modelName T defines a type named T. An abstraction box on the VADT form defines the group of ROs

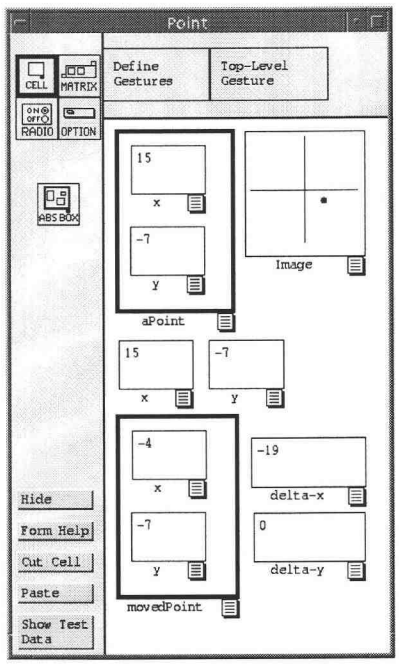

Figure 5.1 The Point form defines two instances of type Point. Abstraction boxes aPoint and movedPoint each contain a value of type Point.

from which an instance of T is constructed, and hence the value of each abstraction box on form T is an instance of type T. For example, in Figure 5.1, the Point form defines the type Point and the abstraction boxes aPoint and movedPoint contain instances of type Point. If a cell X references an abstraction box A, the cell's ROset "virtually" corresponds to the abstraction box's ROset; that is, X's ROset is comprised of virtual ROs, one for each RO in A's ROset. These relationships will be formalized in the next subsections. The notational conventions used in this chapter pertaining to the programming objects are:

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

$\leftarrow$ denotes the referencing operation. For example, X $\leftarrow$ Y means that RO X's formula is a reference to RO Y. (The arrow points in the direction of data flow.)

$\overset{*}{\leftarrow}$ denotes the transitive closure of $\leftarrow$. X $\overset{*}{\leftarrow}$ Z iff either X $\leftarrow$ Z, or X $\leftarrow$ Y and Y $\overset{*}{\leftarrow}$ Z.

$\leftarrow_c$ denotes the constant specification operation. For example, X $\leftarrow_c$ C means that RO X's formula is the constant C.

$\overset{*}{\in}$ denotes the transitive closure of $\in$. X $\overset{*}{\in}$ Z is true iff either X $\in$ Z, or X $\in$ Y and Y $\overset{*}{\in}$ Z.

5.2.2 Formula syntax and semantics

As Table 5.1 shows, there are only three operators in Core Forms/3: the implicit operator referencing another RO ("$\leftarrow$"), the implicit operator specifying equality to a constant ("$\leftarrow_c$"), and the explicit operator "compositionOfParts". Table 5.2 defines the semantics of each of these operators in terms of their preconditions and postconditions. Note that there are no arbitrary input values. As in most spreadsheets, all inputs are constants entered into the program via formula edits, which will be significant to our ability to apply *static* type checking to every part of the program.

formula	::=	*compositionOfParts* | expr
expr	::=	*constant* | ref
ref	::=	RORef | formRef : RORef
formRef	::=	*formID* | *modelName* defSet
defSet	::=	() | (defs)
defs	::=	def | def , defs
def	::=	RORef ≡ expr
RORef	::=	*cellID* | *matrixID* | *matrixID* [subscripts] | *absID* | *absID* [*cellID*] | *absID* [*matrixID*] | *absID* [*matrixID*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

Table 5.1 The grammar for Core Forms/3's formula language. (To minimize the amount of new notation, we also use the terms established here in the formal presentation of the model.) The divider separates the operator syntax from the ref operand syntax. Note that the defSet notation describes formulas that are unique to the form described, for example, the notation if(ifB≡100) indicates a copy of form "if" on which cell ifB contains the constant formula 100.

Formula for X	Preconditions	Postconditions
← Y, where Y is an RO	Y is an existing RO. $Y \not\leftarrow X$. Y is a matrix iff X is a matrix.	X.value = Y.value. |X.ROset| = |Y.ROset|. $\forall Y_i \in$ Y.ROset $\exists X_i \in$ X.ROset, $X_j \leftarrow Y_j$.
$\leftarrow_c$ C, where C is a constant	X is not a matrix.	X.value = C.
compositionOfParts	X is a cell group.	X.value = $\{X_i$.value | $X_i \in$ X.ROset$\}$.

Table 5.2 Axiomatic semantics for each formula possibility in Core Forms/3. Implicit in the notion of equality is the principle that if value1 = value2, then their types are equal. Also note that the "$Y \not\leftarrow X$" precondition prevents circular references. All the preconditions are easily checked statically, and in full Forms/3 are enforced by the environment.

5.2.3 Forms

The above set of operators may seem too limited to get any computation done; for example, there are no conditional or arithmetic operators. However, the functionality normally found in these built-in operators is provided instead by built-in forms named "if", "+", and so on. These forms can be copied as many times as needed to get the desired number of instances, in which case all copies will have the same modelName. Some of the ROs on them have modifiable formulas, and these formulas can be set up to be references to other ROs in the program. The result ROs can then be referenced by other ROs in the program to propagate the results where needed.

The semantics of each built-in form is defined through preconditions and postconditions. Table 3 gives the semantics for one of these forms, although we will expand the expression of type information about such forms later in this paper.

ModelName	ROset	Preconditions	Postconditions
+	contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC.	plusA.value is a number. plusB.value is a number.	plusC.value is a number that is the sum of plusA.value and plusB.value.

Table 5.3 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 there is no state modification in Core Forms/3 (or in Forms/3), the preconditions are invariant. Form + is one of the forms built into Core Forms/3 to implement a primitive operation; the others are: -, *, /, mod, =, and, or, >, <, not, width, height, if, and compose. (See [Burnett 1991] for details of all primitive forms.)

As Table 5.3 implies, all copies of + have the "same" cells on them, although their formulas may be different. More precisely (and more generally), this relationship is described as the first property [Copies] in Table 5.4, which says that forms with the same modelName always have the same cellIDs in their ROsets.

[Copies] $\dfrac{\text{F and G are forms, F.modelName = G.modelName}}{|F.ROset| = |G.ROset|,\ \forall\ f \in F.ROset,\ \exists\ g \in G.ROset \text{ such that } g.cellID = f.cellID}$

[VADTformExistsC] $\dfrac{\text{C is of type T, where C is a constant}}{\exists \text{ VADT form } T_C = T(MainAbs \equiv C),\ T_C.modelName = T}$

[VADTformExistsR] $\dfrac{\text{X.value is of type T, where X is an RO}}{\exists \text{ VADT form } T_X = T(MainAbs \equiv X),\ T_X.modelName = T}$

[Inst] $\dfrac{\text{X.value is of type T, where X is a cell}}{X \overset{*}{\leftarrow} F{:}Y, \text{ where Y is an abstraction box and F.modelName = T}}$

[AbsType] $\dfrac{\text{T:X is an abstraction box}}{\text{X.value is of type } T_X.modelName = T.modelName}$

[AbsStruc] $\dfrac{\text{T:X, T:Y are abstraction boxes, } X_i \in X.ROset}{|Y.ROset| = |X.ROset|,\ Y_i \in Y.ROset,\ Y_i \text{ is a matrix iff } X_i \text{ is a matrix}}$

Table 5.4 Properties of forms.

Recall from Section 1.2 that VADT forms can be used to define new types, and all built-in types are also described with these forms. This property stated formally in Table 5.4. The only difference between user-defined types and built-in types is whether some of the formulas for the ROs on the VADT form had to be created by the language implementor. One abstraction box on each VADT form is distinguished by having cellID "MainAbs". (In full Forms/3, ROs have an optional user-defined name attribute which the programmer can use in place of the ID; hence in Figure 5.1, the RO with cellID MainAbs appears on the screen with name "aPoint".) Although there can be additional abstraction boxes, all abstraction boxes on the same form have ROsets of the same size and structure and their values have the same type (properties [AbsType] and AbsStruct]). For every RO X, there is a VADT form denoted T_X whose main abstraction box references X such that

X.value is of type T ([VADTformExsitsC] and [VADTformExistsR]; [Inst] follows from these). Further, each T_I has the same modelName as VADT form T. Hence a constant formula ($\leftarrow_c$ C) can always be replaced by a reference to T_C:MainAbs.

5.2.4 Translating between Forms/3 and Core Forms/3

Core Forms/3 includes all the programming objects of Forms/3 except radio cells and popup-menu cells (such as those seen on the built-in form circle from Figure 1.1). The translation of full Forms/3 programming objects to Core Forms/3 programming objects consists of removing their cosmetic attributes (such as positions, borders, cellnames distinct from cellIDs, etc.) Since radio cells and popup-menu cells are only cosmetically different from ordinary cells, eliminating cosmetics effectively reduces them to ordinary cells.

However, one cosmetic attribute, position, is semantically significant in Forms/3 in one situation: if RO X is positioned inside another RO Y, then not only is X an element of Y.ROset, but also Y.formula is implicitly defined to be compositionOfParts. Core Forms/3 makes this formula explicit: if the Forms/3 user has not given Y an explicit formula, the translation to Core Forms/3 explicitly defines Y.formula to be "compositionOfParts".

All other formulas in Forms/3 are explicit. Explicit formulas expressed by direct manipulation and gestures are translated to Core Forms/3 in two steps: first by translating them to a series of ordinary textual Forms/3 formulas (using equivalents presented in detail in [Burnett and Gottfried 1998]), and then proceeding as below.

Since operands are the same in Forms/3 as in Core Forms/3, only the operators need to be translated. Translating a Forms/3 operator in a formula for RO X is done by expressing X.formula as a reference to an RO on another form. Consider a formula that contains only one operator. Hence there are no subexpressions, and each argument is an

expr (following Table 5.1). If the prefix version of X.formula is op $expr_1$, $expr_2$, ..., then X.formula in Core Forms/3 is:

$$\leftarrow op(Arg1 \equiv expr_1, Arg2 \equiv expr_2, ...):Result$$

where Arg1, Arg2, ... are modifiable ROs on form op, and Result is an unmodifiable RO containing the result of the built-in operation. (The above notation is as defined for defSet in Table 5.1.) For Forms/3 formulas with embedded subexpressions, each embedded subexpression is replaced (bottom up) by a reference following the above translation scheme; this results in the parent expression having arguments of the required syntax, so that translation can proceed up another level of the tree, and so on to the root. See Figure 5.2 for an example.

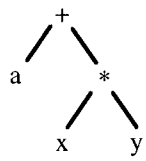

Figure 5.2 Translation from Forms/3 to Core Forms/3. Translation of the formula "(a + (x * y))" proceeds bottom-up, first translating (x*y) to *(timesA ≡ x, timesB ≡ y):timesC, and then incorporating that translation into the full expression, resulting in the Core Forms/3 formula being a reference to +(plusA ≡ a, plusB ≡ *(timesA ≡ x, timesB ≡ y):timesC):plusC.

Since all of Core Forms/3 except the explicit operator compositionOfParts is a legal subset of Forms/3 (assuming some default set of cosmetic attributes), then the only translation mechanism needed to translate Core Forms/3 to Forms/3 is to deal with compositionOfParts, making the formula implicit in the Forms/3 programming object Y by positioning all elements of Y.ROset within the bounding box of Y.

5.3 Polymorphism

An important feature of many approaches to static type inference is polymorphism, or the ability to infer more than one type for a single program entity. Without this flexibility, static typing would be much more rigid, requiring the duplication of certain pieces of code in order to accommodate a variety of types. There might, for example, need to be two separate plus operations, one defined for integers and another for real numbers.

This section provides a very brief introduction to the basic ideas of polymorphic type inference. For more comprehensive coverage, two excellent introductory surveys are [Cardelli and Wegner 1985] and [Schwartzbach 1997]. We then give examples of the kind of polymorphism we want our type system to allow.

5.3.1 Basic concepts of static polymorphic type inference

The term *polymorphic types* refers to "data or programs which have many types or operate on many types" [Cardelli 1987]. For a programming language that requires explicit type declarations, a polymorphic variable X may be declared as:

var X: α

where α is a type variable whose actual meaning varies contextually. This explicit approach is referred to as *explicit polymorphism*. The term *implicit polymorphism* is used to describe polymorphic approaches in which such type declarations are unnecessary. In these cases, type information is automatically inferred by the language processor. Most inferences are made statically. The goal is to preserve *type safety*, that is, if a program is statically determined to be type-safe, then the type system guarantees that no run-time type errors will arise.

Most type inference systems include function types which allow languages with higher-order functions to employ type inference. Declarative languages with higher-order

functions are the class of languages in which type inference is most commonly found. For example, suppose in such a language, the following function has been defined (adapted from [Jun and Michaelson 1998]):

```
fun     sumfunc _ [ ] = 0 |
        sumfunc f (h::t) = f  h + sumfunc f t
```

The function sumfunc sums the result of applying the function f to every element of a list. It has type (α ➜ int) ➜ α list ➜ int, where α is a polymorphic type, and the arrows separate the arguments and the return value in a function's type (e.g., α ➜ int means a function taking a polymorphic type α and returning an integer). Thus as its first argument, sumfunc can take any function of one argument that returns an integer (with the additional caveat that the one argument must be the same type as the items in the list argument to sumfunc). Suppose two additional functions, cardinal and square, are defined with the following types:

cardinal: int ➜ string
square: int ➜ int

If cardinal is passed into sumfunc as the first argument (f), then a type error would occur because the system could not resolve the type conflict between int and string. If square is used, however, no type error would arise, and the following type schemes may result:

sumfunc: (α ➜ int) ➜ α list ➜ int	by initial definition
square: int ➜ int	by initial definition
sumfunc: (α ➜ int) ➜ α list ➜ int ⇒ α = int	by substituting square for f
sumfunc: (int ➜ int) ➜ int list ➜ int	final result for this use of sumfunc

In the same way, sumfunc could be used with function truncate of type real ➜ int or function length of type string ➜ int. Such reusability is the benefit of polymorphism.

The sumfunc function is an example of *operation polymorphism* [Blair et al. 1989] because it is code that works on different types. This is in contrast to the kind of polymorphism generally associated with object-oriented languages, *inclusion*

polymorphism, which instead allows reuse based on an object belonging to more than one type.

5.3.2 Polymorphic programming in Forms/3

Our approach to polymorphism does not use inclusion polymorphism; each value belongs to exactly one type even in the presence of similarity inheritance. Instead, it uses operation polymorphism. Any operation on a user-defined form can be used polymorphically. The programmer creates a polymorphic operation in Forms/3 by referencing some example of a type that the operation operates on and, if warranted, the system generalizes the concrete references automatically to make the operation polymorphic. (Hence, the programmer need not plan in advance to use the operation polymorphically.) This generalization for polymorphism is an extension of the generalization of [Yang and Burnett 1994]. Without generalization for polymorphism, similarity inheritance would result in reuse of code only via inheritance—procedural abstractions could not be used on types of input different from those of the original sample values. After generalization, however, a procedural abstraction can work on a number of different types in a way similar to the dynamic dispatching of object-oriented languages. IT is possible to apply static type inference to a dynamic dispatch situation because there is no arbitrary run-time input, as mentioned in Section 5.2.2. The reason static type inference is preferable to execution even when the inputs are available is that it is more general and can detect errors that a single test run would not find.

Figure 5.3 shows a simple example (in full Forms/3) of how our extended generalization allows operation polymorphism. The Mirror form has three cells: the cell oldPoint contains a value of type Point; the cell x-offset contains a number indicating the desired change for the point's x value; the cell newPoint references cell movedPoint on a copy of the form Point (form 248-Point in this case). All of these formulas are concrete

(referring to cells on specific copies of forms) and therefore monomorphic. Generalization for polymorphism is only done when the system detects a type emergency[1]. When the programmer makes a new copy of the form Mirror (500-Mirror in the figure) and changes the formula of cell 500-Mirror:oldPoint so that it contains a value of type ColorPoint, such an emergency occurs. If the inherited formula of cell 500-Mirror:newPoint were used in its concrete form, the system would incorrectly generate a copy of the form Point and attempt to set the formula of its abstraction box MainAbs to reference cell 500-Mirror:newPoint. Since abstraction boxes are only allowed to accept values of their monomorphic type (defined by the name of their model form), an abstraction box on any Point form can only accept values of type Point (this will be formalized in Section 5.4) and an error would be raised. Instead, generalization is triggered and a copy of the ColorPoint form—rather than the Point form—is generated, allowing the Mirror form to perform correctly for its new input. Intuitively what happens is the concrete reference to "the copy of Point named 248-Point" is generalized to "the VADT form whose abstraction box MainAbs references cell oldPoint and whose cell delta-x references cell x-offset." In the notation of this chapter, the generalized formula for 500-Mirror:newPoint is "$F_{oldPoint}$(aPoint≡500-Mirror:oldPoint, delta-x≡500-Mirror:x-offset):movedPoint." Because it can be (or has been) generalized for polymorphism, we will refer to a formula that references a VADT form as a *polymorphic reference* if the main abstraction box's formula is not *compositionOfParts*. The programmer's view uses a legend notation (extended from [Yang 1996]) to represent the generalized formula, as shown in the figure.

---

[1]Yang's approach included several triggers, including saving a form or removing it from the screen's display, because information needed for generalization would be lost otherwise. In the case of generalization for polymorphism, all required information is always available in the concrete formula, so only one trigger is necessary.

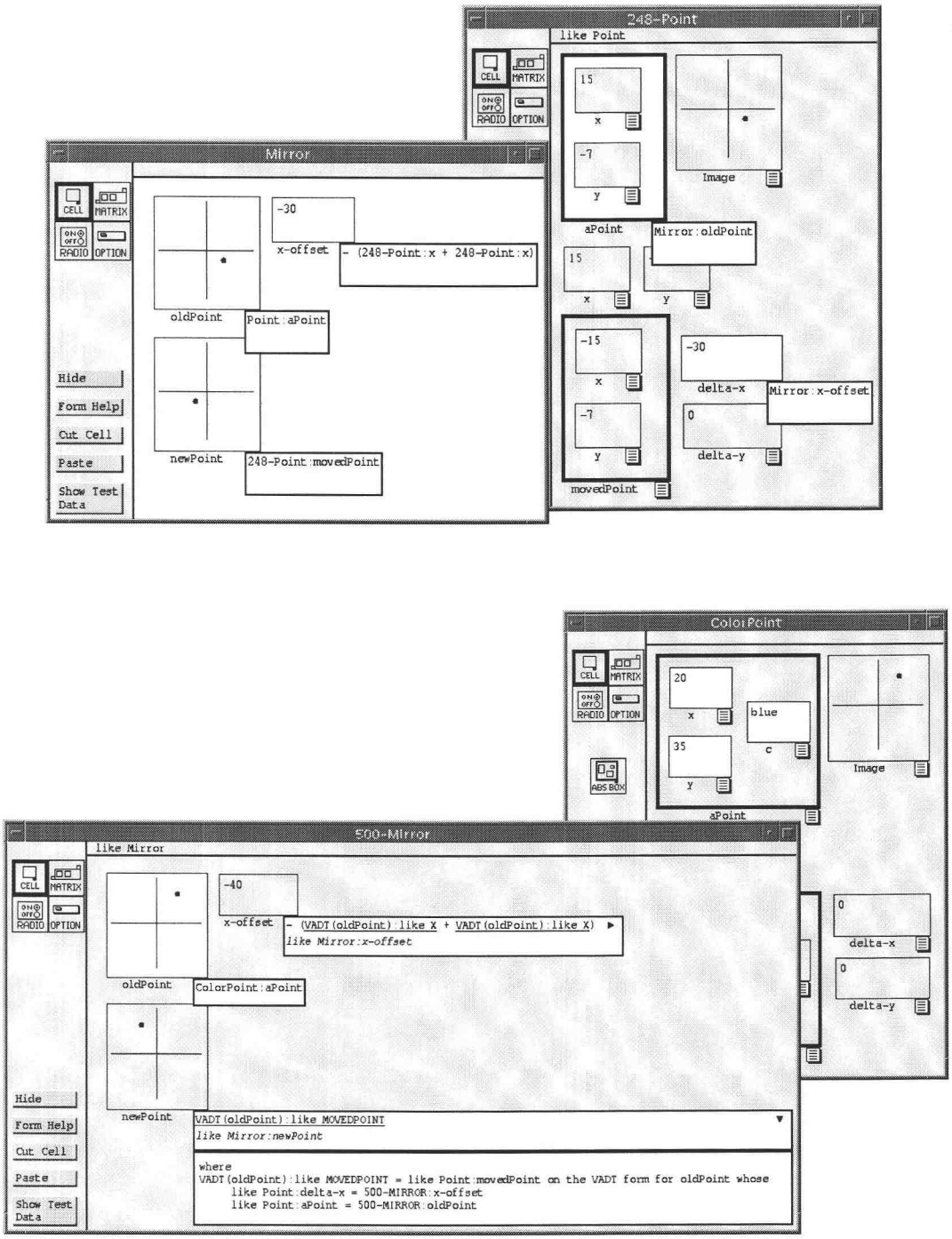

Figure 5.3 An example of operation polymorphism. (Top) The form Mirror as it was first programmed. The references to the copy of form Point are concrete. (Bottom) A copy of form Mirror after generalization for polymorphism. The formula for newPoint is shown with its legend visible.

In general, the notation "X ← $F_{G:Y}$(Arg1≡$expr_1$, Arg2≡$expr_2$, ...):Op" reflects the results of generalization for polymorphism, and means that X references Op on the VADT form F whose main abstraction box references G:Y[1]. Thus, if the type of Y.value varies on different copies of form G, such as if Y is only one example of the elements of some matrix that must all be processed, then so does the selection of which VADT form's Op to reference.

The example in Figure 5.3 illustrates operation polymorphism through the reuse of a form. Another way to use polymorphism in Forms/3 is with a heterogeneous matrix. For example, suppose the programmer filled a matrix named *collections* with stacks, queues, and other collections, then used another matrix to reference the sizes of the collections. In Forms/3 the second matrix would have one general formula to be used for each subcell in the matrix; in this case "the size cell on the VADT form for collections[i@j]" where i and j are used to indicate the corresponding subcell of the collections matrix.

Theoretically, it would also be possible to use polymorphism over time. Forms/3 has an explicit notion of time in which each formula actually defines a stream-like collection of values rather than a single atomic value. However, in order to simplify the type system and notation, our approach assumes homogeneous streams. Thus reasoning about streams can be done as a unit, in the same manner as if each formula did simply define an atomic value. As a result, we will ignore time in the rest of the discussion of types.

---

[1] The defSet is necessary here because some cells Argi on the generalized instances of F may have formulas of $expr_i$. This information is part of the system's generalization algorithm—from the user's perspective the polymorphic reference is the same as a regular reference.

5.4 Model of Types

5.4.1 Fine-grained reasoning in terms of guarantees versus requirements

One goal of our model of types is that it be comprehensible 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. (We emphasize that we are speaking of a model of types being usable in these two different sorts of VPLs, not of a single VPL being usable by these two audiences.) In order to achieve the first goal without sacrificing the second, we needed to try an approach that departs from traditional approaches, reasoning at a very fine-grained level about individual operations guaranteed and required rather than about entire types as atomic units. Working at this granularity eliminates the need to reintroduce declarations of interfaces or subtype relationships, which would have run counter to the goal of potential use by end users. This need to eliminate reliance on such programmer-oriented concepts is one of the main reasons a model of types different from those developed for textual languages is needed for use by VPLs aimed at end users.

Hence, in our model, there are no subtypes, compositions of types, or interfaces. Further, type checking is not done based on types as atomic units; rather the sets of operations guaranteed by each RO X and required by formulas referring to X are compared. If the requirements are not a subset of the guarantees, then there is a type error. For example, if formulas reference the "grow" and "shrink" operations on cell X's VADT form then X can contain any type of value that provides the "grow" and "shrink" operations. The sets of required and guaranteed operations are statically inferred by the type system. The set of operations guaranteed by X are denoted G(X), and the set of

operations required of X are denoted R(X). In this new model of types, type safety is defined as follows:

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

5.4.2 Guarantee sets

In general, each RO guarantees all the operations defined on its VADT form. Since in Core Forms/3, each operation is associated with an RO on the VADT form, each operation 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 identify their type. Thus the guarantee set for an abstraction box F:A is the collection of operations available on that same (VADT) form F. Since operations are cellIDs, the axiom is:

[GA] $G(A) = \{x \mid x \overset{*}{\in} F.ROset\}$ where A is an abstraction box on form F

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: the operations radius, thickness, lineStyle, lineForeColor, etc., including the abstraction box newCircle (refer back to Figure 1.1 to view primitiveCircle). Other circle-related tasks that can be performed using these operations do not need to be included on the primitiveCircle form itself. In this paper, we abbreviate the set of low-level operations for these primitive types (the cellIDs on their VADT forms) as "<primitiveType>Operations"; in this example "primitiveCircleOperations".

ROs with constant formulas simply derive their guarantee set from the primitive form describing the constant's value (see Table 5.5).

[GC] $\dfrac{X \leftarrow_c C}{G(X)=G(C)}$ where C is a constant where $G(C) = \{y \mid y \overset{*}{\in} F_C.ROset\}$

Example constants C	G(C)
C = 3	numberOperations
C = "hello"	textOperations
C = true	booleanOperations

Table 5.5 Every constant value guarantees exactly the operations on its primitive form. For succinctness, we name these sets <primitiveType>Operations rather than listing the individual cellIDs.

For cells and matrices with a formula whose operator is "←", the guarantee set simply propagates from the referenced cell or matrix by the axiom below. The "where" clause restricts this axiom to cells and matrices only because it is not needed for abstraction boxes; they are already handled by axiom [GA]. However, removing this restriction would not cause any adverse effects, since it would not introduce any conflicts with [GA].

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

The above three axioms handle every legal Core Forms/3 formula except matrices with compositionOfParts formulas. In this case, 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 since the resulting guarantee set will be the same whether [Gref] or [GM] is applied to a matrix with a reference ("←") formula.

[GM] $G(M) = \cap\, G(M[i])$ for all $M[i] \in M.gridROset$

[GM] is the first of several elements of this presentation that are different for matrices than for other ROs. It would have been possible to change the language definition of matrices to make them define a complex value, as do abstraction boxes, and this would eliminate most of the specialized matrix reasoning. We elected not to do so because reasoning about Core Forms/3 matrices show how the model can apply 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 5.6.

ModelName	ROset	Type-related postconditions: Guarantees
+	contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC.	G(plusC) = numberOperations
>	contains modifiable cells with cellIDs greaterthanA, greaterthanB, and unmodifiable cell with cellID greaterthanC.	G(greaterthanC) = booleanOperations
append	contains modifiable matrices with cellIDs appA, appB, and unmodifiable matrix with cellID appC.	G(appC) = G(appA) ∩ G(appB)
if	contains modifiable cells with cellIDs ifA, ifB, ifC, and unmodifiable cell with cellID ifD.	G(ifD) = G(ifB) ∩ G(ifC)

Table 5.6 Guarantee sets for some primitive forms. For the primitive forms implementing operations, 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 3). Preconditions for these forms will be given in the next subsection.

If the guarantee set of a cell or cell group is empty, the cell or cell group guarantees only universally polymorphic primitive operations, such as height and width.

5.4.3 Requirement sets

Although 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 propagate against dataflow, because they indicate how the RO is to be used by other parts of the program. Further, since there may be many parts of the program that make use of (reference) a single RO, all of these uses must be collected. The implication of these two 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.

There are two ways an operator Op can become part of R(Y): either because Op $\in$ R(X) and X references Y (X $\leftarrow$ Y), or because some RO Z contains a polymorphic reference to Op on Y's VADT form (Z $\leftarrow$ F_Y:Op). The following axiom combines these to determine the requirement set for Y. The multiple $F_Y(defSet_i)$ defSets allow for multiple copies of a VADT form whose main abstraction boxes reference Y, which could occur if some of the ROs on the form need new formulas to provide appropriate "parameters" for the referenced Op_i. So, in addition, the axiom requires that these parameters (the ROs mentioned in the defSets) exist.

[R1] $$\frac{X_1, X_2,...X_n \leftarrow Y \text{ and } Z_1 \leftarrow F_Y(defSet_1):Op_1,\ Z_2 \leftarrow F_Y(defSet_2):Op_2...Z_m \leftarrow F_Y(defSet_m):Op_m}{R(Y) = \cup\ i=1..n \text{ of } R(X_i) \cup \{Op_1, Op_2,...Op_m\} \cup k=1..m\ \{Op \mid Op \in defSet_k\}}$$

where X_i is not an abstraction box and each $defSet_k$ is "$Arg_{k1}\equiv expr_{k1}$, $Arg_{k2}\equiv expr_{k2}$, ..." In addition, the lists of X's and Z's are complete, that is, if X is not an abstraction box and X $\leftarrow$ Y then X $\in$ $\{X_1, X_2,...X_n\}$, and similarly for the Z's.

Numrows and numcols cells are always known to require numberOperations.

[RN] $$R(N) = numberOperations$$

Matrices require everything that the cells in their gridROsets require:

[RM] $R(M) = \cup R(M[i])$ where $M[i] \in M.gridROset$

Preconditions on ROs on the primitive forms add to the requirements propagating through the system; see Table 5.7.

ModelName	ROset	Type-related preconditions: Requirements
+	contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC.	R(plusA) = numberOperations R(plusB) = numberOperations
>	contains modifiable cells with cellIDs greaterthanA, greaterthanB, and unmodifiable cell with cellID greaterthanC.	R(greaterthanA) = numberOperations R(greaterthanB) = numberOperations
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)[1] 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) = booleanOperations 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 5.7 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 postconditions 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 5.6.)

[1]Or, more formally,

$$\frac{X_1, X_2,...X_n \leftarrow Y \text{ and } Z_1 \leftarrow F_Y(defSet_1):Op_1, Z_2 \leftarrow F_Y(defSet_2):Op_2...Z_m \leftarrow F_Y(defSet_m):Op_m}{R(appA) = \cup\, i=1..n \text{ of } R(X_i) \cup \{Op_1, Op_2,...Op_m\} \cup k=1..m \{Op \mid Op \in defSet_k\} \cup R(appC)}$$

5.4.4 Recursion

Although semantically valid recursive calls in Core Forms/3 eventually terminate, the point of termination is not statically known. These 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 must have the same guarantee as the non-recursive branch.

[RecB] $\dfrac{F{:}X \leftarrow \text{if(ifA}\equiv\text{Adef,ifB}\equiv\text{Bdef,ifC}\equiv\text{Cdef):ifD, F':X} \overset{*}{\in} \text{Bdef}}{G(F{:}X) = G(\text{ifA}\equiv\text{Adef,ifB}\equiv\text{Bdef,ifC}\equiv\text{Cdef):ifC})}$ where F.modelName = F'.modelName

[RecC] $\dfrac{F{:}X \leftarrow \text{if(ifA}\equiv\text{Adef,ifB}\equiv\text{Bdef,ifC}\equiv\text{Cdef):ifD, F':X} \overset{*}{\in} \text{Cdef}}{G(F{:}X) = G(\text{ifA}\equiv\text{Adef,ifB}\equiv\text{Bdef,ifC}\equiv\text{Cdef):ifB})}$ where F.modelName = F'.modelName

5.4.5 How similarity inheritance fits into the type inference model

To add support for polymorphism based on similarity inheritance to the basic axioms already presented, it suffices to supplement one guarantee axiom. Recall that "$\rightarrow$" is the "shares with" relation introduced in the previous chapter.

[GA'] $G(A) = \{x, \text{like } y \mid x \overset{*}{\in} F.\text{ROset}, y \overset{*}{\rightarrow} x\}$ where A is an abstraction box on form F

This allows more programs to be inferred to be type safe, since the guarantee sets are larger than before "$\rightarrow$" relationships were present. The definition of type safe also needs to be adjusted; under similarity inheritance, a program is now type safe if the guarantees set for every RO contains either "Op" or "like Op" for every operation Op in its requires set.

Revised Definition: $\forall X \overset{*}{\in}$ program P and $\forall$ Op $\in$ R(X), if either Op $\in$ G(X) or "like Op" $\in$ G(X), then P is *type safe*.

Thus a type error is now defined as: $\exists$ X such that Op $\in$ R(X), and Op $\notin$ G(X) and "like Op" $\notin$ G(X).

5.5 Examples of Type Inference in Forms/3

In this section, we present several examples of the guarantee and requirement sets that can be inferred. We use full Forms/3 because it is a real environment and thus allows actual screenshots. Although the axiom set is given for Core Forms/3, formal reasoning about full Forms/3 is possible using the translations between Forms/3 and Core Forms/3 specified in Section 5.2.4.

5.5.1 Example: Type inference without inheritance

The form in Figure 5.4 is an example of a program with no inheritance. The population program provides different sized circles to represent cities depending on their population. The types for each RO on the form population can be inferred using the axioms provided in Section 5.4. For conciseness, we omit form names where doing so does not introduce ambiguity.

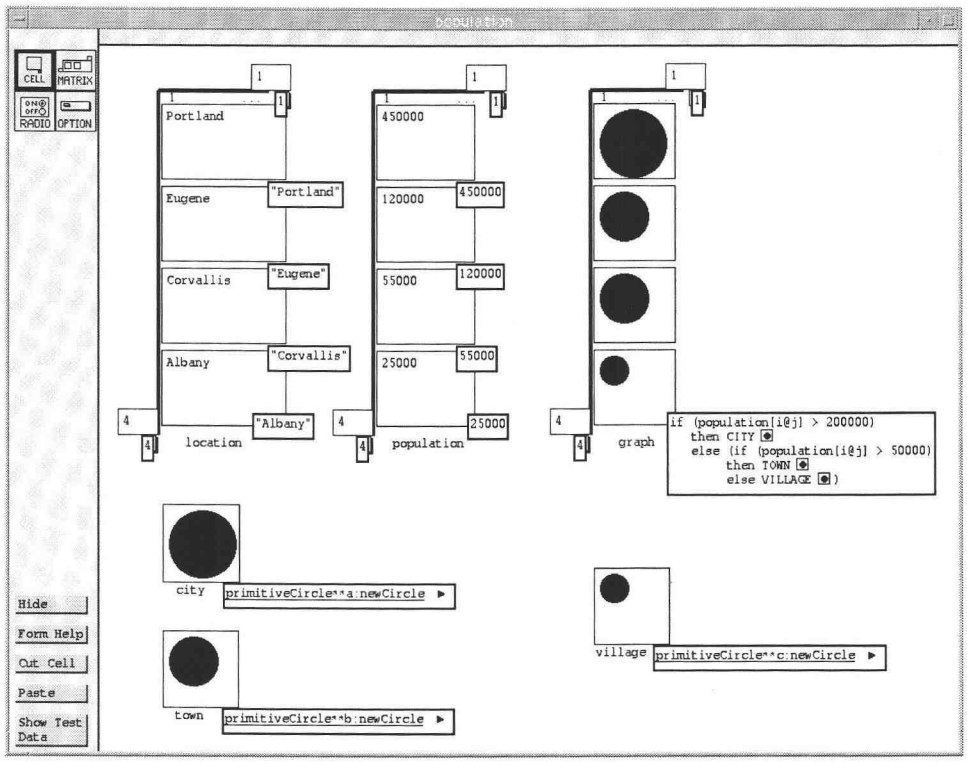

Figure 5.4 The population program.

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

$$R(location[1@1]) = \{\} \qquad [R1]$$
$$G(location[1@1]) = G(\text{“Portland”}) = textOperations \qquad [GC]$$

Obviously, $R(location[1@1]) \subseteq G(location[1@1])$, so there is no type error 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 since city is referenced by other ROs in the program. Here the formula for the subcells in the graph matrix is first translated to the Core Forms/3 equivalent, so, for example, there is a copy of the if form on which ifB.formula = "← population:city".

R(city) = R(ifB)	[R1]
= R(ifD)	[Table 5.7]
= R(graph[1@1]) ∪ R(graph[2@1]) ∪ R(graph[3@1]) ∪ R(graph[4@1])	[R1]
= { }	[R1]
G(city) = G(702-primitiveCircle:newCircle) = primitiveCircleOperations	[GC]
∴ R(city) = { } ⊆ primitiveCircleOperations = G(city)	

The town and village cells have similar derivations with the same results.

The matrix subcell graph[1@1]'s formula is a reference rather than a constant. Again translating the formula for the subcells in the graph matrix to the Core Forms/3 equivalent, let if1 and if2 be the appropriate copies of "if"; for example, if2:C ← village, if1:C ← if2:ifD, and so on. Then:

R(graph[1@1]) = { }	[R1]
G(graph[1@1]) = G(if1:ifD)	[Gref]
= G(city) ∩ G(if2:ifD)	[Table 5.6]
= G(city) ∩ (G(town) ∩ G(village))	[Table 5.6]
= primitiveCircleOperations	[GC]
R(graph[1@1]) = { }	[R1]
∴ R(graph[1@1]) = { } ⊆ primitiveCircleOperations = G(graph[1@1])	

The same results apply for the other cells in graph's gridROset.

Turning to a cell with a non-empty requirement set, the population[1@1] cell has a requirement set of numberOperations. (Forms ">1" and ">2" are copies of form ">" on which greaterthanA ← population[1@1].)

R(population[1@1]) = R(>1:greaterthanA) ∪ R(>2:greaterthanA)	[R1]
= numberOperations	[Table 5.7]
G(population[1@1]) = G(450000)	[GC]

= numberOperations
∴ R(population[1@1]) = numberOperations ⊆ numberOperations = G(population[1@1])

The same results apply to the other cells in population's gridROset. The population matrix itself 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 its subcells.

R(population) = R(population[1@1]) ∪ R(population[2@1]) ∪ R(population[3@1]) ∪
R(population[4@1]) [RM]
= numberOperations [Table 5.7]
G(population) = G(population[1@1]) ∩ G(population[2@1]) ∩ G(population[3@1]) ∩
G(population[4@1]) [GM]
= numberOperations [GC]
∴ R(population) = numberOperations ⊆ numberOperations = G(population)

In the same way, the matrix location can be shown to have an empty requirement set and a guarantee set of textOperations and the matrix graph can be shown to have an empty requirement set and a guarantee set of primitiveCircleOperations.

Each of the three matrices on the population form also has a numrows cell and a numcols cell in its ROset. In this program, they happen to have the same requirement and guarantee sets, so we give only one example here.

R(location[numrows]) = numberOperations [RN]
G(location[numrows]) = G(4) = numberOperations [GC]
∴ R(location[numrows]) = numberOperations ⊆ numberOperations = G(location[numrows])

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

5.5.2 Example: Type inference in the presence of single inheritance

The Stack and Queue examples introduced in Chapter 4 illustrate how our model of types works in the presence of inheritance. First consider what the main abstraction boxes on each form guarantee. These sets are lengthy, because they include the cellIDs of every RO $\overset{*}{\in}$ the ROset for the forms. (Note that type safety is separated from information hiding, which is solved in a different way [Burnett and Ambler 1994].) Although shared data structures in a system implementing this model can cut down on some duplication of these lengthy sets, not all copying can be avoided. (Refer back to Figures 1.2 and 4.1 to see the Stack and Queue forms.)

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

G(Queue:Queue) = {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], ...} [GA']

Since Queue was created via similarity inheritance from Stack, the guarantee sets for abstraction boxes on a Queue form include shared ("like") operations. Notice, however, that it does not include "like push" because the programmer overrode the similarity between Stack's push and Queue's enqueue.

Figure 5.5 shows a simple form illustrating some uses of operations on a stack. The cell collection references a Stack, so its guarantees set is the same as G(Stack:Stack). Assuming that cells removed-item and the-rest are the only polymorphic references to cell collection's VADT form, cell collection's requirements set contains two operations:

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

Since G(Queue) includes "like top" and "like pop," a reference to Queue would also result in a valid value for the cell collection according to the revised definition of type safety in Section 5.4.

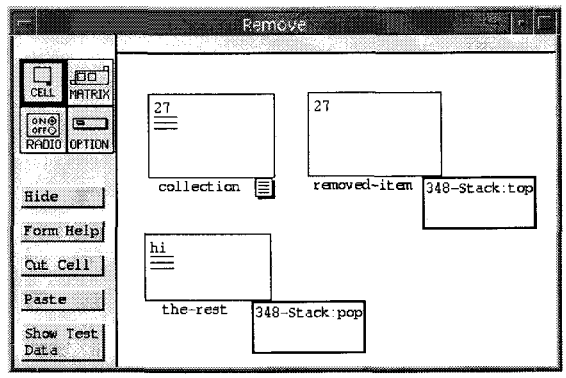

Figure 5.5 Polymorphic references. The form Remove contains simple examples of polymorphic references in cells removed-item and the-rest. These are shown in their concrete form (they have not been generalized).

5.5.3 Example: Type inference in the presence of multiple inheritance

The Deque (double-ended queue) in progress from Figure 4.2 illustrates use 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.

5.6 Discussion

This section analyzes properties of the model of types, first showing soundness and completeness of the axiom set, then investigating the potential understandability of type errors and finally examining the types of Forms/3's equivalent to function arguments.

5.6.1 Soundness

Let **A** be the axiom set presented in Section 5.4. In the context of this type system, our interest is in whether implementing **A** in a VPL will result in soundness with regard to type safety, ensuring that the VPL's users are informed of every type error. In order to be sound, a program with type errors cannot be proven type safe using **A**. For the purpose of proving soundness, we define a *use* of Op on X as a (polymorphic) reference to Op on X's VADT form.

The model will be said to be sound with respect to guarantees if G(X) contains only operations guaranteed to be on X's VADT form, that is:

(a) where X is not a matrix: If Op $\in$ G(X) according to **A**, then every use of Op on X will be successful (Op $\in$ F_X.ROset) and if "like Op" $\in$ G(X) according to **A**, then every use of Op on X is defined as successful use of "like Op" on X (Op2 $\in$ F_X.ROset where Op $\xrightarrow{*}$ Op2).

(b) where X is a matrix: If Op $\in$ G(X) according to **A**, then every use of Op on X[i] will be successful (Op $\in$ $F_{X[i]}$.ROset) and if "like Op" $\in$ G(X) according to **A**, then every use of Op on X[i] is defined as successful use of "like Op" on X[i] (Op2 $\in$ $F_{X[i]}$.ROset where Op $\xrightarrow{*}$ Op2).

Proof:
(a) X is not a matrix
Case 1: X is an abstraction box.
$Op \in G(X) \Rightarrow Op \overset{*}{\in} F.ROset$ (where F:X is X) by [GA']
$F_X.modelName = F.modelName$ by [AbsType]
$\Rightarrow Op \overset{*}{\in} F_X.ROset$ by [Copies]
$\Rightarrow$ every use of Op on X will be successful.
By similar reasoning, "like Op" $\in G(X) \Rightarrow$ every use of Op on X will be successful.
Case 2: $X \overset{*}{\leftarrow} F:Y$, where F:Y is an abstraction box. by [Inst]
$G(X) = G(F:Y)$ by (repeated application of) [Gref]
$Op \in G(X) \Rightarrow Op \in G(F:Y)$
$\Rightarrow Op \overset{*}{\in} F.ROset$ by [GA']
$\Rightarrow F_X.modelName = F.modelName$ by [AbsType] and [VADTformExistsR]
$\Rightarrow Op \overset{*}{\in} F_X.ROset$ by [Copies]
$\Rightarrow$ every use of Op on X will be successful.
Again by similar reasoning, "like Op" $\in G(X) \Rightarrow$ every use of Op on X will be successful.
Case 3: $X \leftarrow_c C$.
Because constants are shortcuts for referencing an abstraction box on a built in VADT form, the proof can be reduced to case 2.
(b) X is a matrix.
$Op \in G(X) \Rightarrow Op \in G(X[i])\ \forall\ X[i] \in X.gridROset$ by [GM]
$\Rightarrow Op \overset{*}{\in} F_{X[i]}.ROset\ \forall\ X[i] \in X.ROset$ by (a) above
$\Rightarrow$ every use of Op on X[i] will be successful.
By the same reasoning, "like Op" $\in G(X) \Rightarrow$ every use of Op on X will be successful.

The model will be said to be sound with respect to requirements if R(X) contains every operation required of X, that is:

(a) where X is not a matrix: If $Op \notin R(X)$ according to **A**, then there is no use of Op on X ($\nexists$ Z such that $Z \leftarrow F_X:Op$) and there is no use of Op on some Y such that $Y \overset{*}{\leftarrow} X$.

(b) where X is a matrix: If $Op \notin R(X)$ according to **A**, then there is no use of Op on any X[i] ($\nexists$ Z such that $Z \leftarrow F_{X[i]}:Op$)

Proof (by contradiction):
(a) X is a cell or abstraction box.
Assume there is a use of Op on X. Then either Case 1 or Case 2 below must be true.
Case 1: $\exists$ Z such that $Z \leftarrow F_X:Op$
$\Rightarrow Op \in R(X)$. by [R1]
But $Op \notin R(X)$, so there is no such Z.
Case 2: There is a use of Op on Y and $Y \overset{*}{\leftarrow} X$
$\Rightarrow \exists$ Z such that $Z \leftarrow F_Y:Op$

$\Rightarrow$ Op $\in$ R(Y) $\Rightarrow$ Op $\in$ R(X). by [R1]
But Op $\notin$ R(X), so there is no such Z.
Thus there is no use of Op on X.
(b) X is a matrix.
Assume there is a use of Op on X[i]
$\Rightarrow \exists$ Z such that Z $\leftarrow F_{X[i]}$:Op
$\Rightarrow$ Op $\in$ R(X[i]) by [R1]
$\Rightarrow$ Op $\in$ R(X). by [RM]
But Op $\notin$ R(X), so there is no such Z.
Thus there is no use of Op on X[i].

The model is not complete because, like other static type systems, it is conservative. A requirements set may contain operations that are not used and a guarantee set may not include all operations on the relevant VADT form. Let R0(Y) be a set of requirements for Y inferred under a model sound and complete with respect to requirements, and let R1(Y) be the requirements inferred using **A**. It is possible for Op to be in R1(Y) when Op is never used for Y. For example, suppose X.formula = "If true=false then (Y div 2) else 3"; this will cause an inference under **A** that Y needs to be dividable, that is, R1(Y) = {div}. Yet, the division will never happen, that is, R0(Y) = { }. However, we have shown that **A** is sound with respect to requirements R1(Y), and from this it is clear that R0(Y) is a subset of R1(Y), both in the case of this example and for any other arbitrary Y. In the presence of soundness, too many requirements produces too conservative a view of which programs are type safe, but it does not allow genuine type errors to be ignored. This is because if a program is inferred to be type safe under **A**, then for any Y in the program, R1(Y) is a subset of G(Y), which means that R0(Y) is also a subset of G(Y). Thus every program inferred to be type safe under **A** would also be inferred to be type safe under a complete system.

Similarly, the model is not complete with respect to guarantees. In the cases in which guarantees are inferred by intersecting other sets of guarantees, the results can be overly conservative. However, because the model is sound with respect to guarantees, by similar reasoning to that for requirements, the lack of completeness results only in an

overly conservative measure of which programs are type safe—it does not allow unsafe programs to be deemed to be safe under **A**.

Hence, the soundness property is sufficient to guarantee that implementing **A** in a VPL will ensure type safety, enabling the VPL to notify its users of every type error in their programs.

5.6.2 Understandability

One of the goals of this work has been to devise a type system understandable enough that it could be used in end-user languages as well as languages for programmers. Although the question of understandability by a particular audience ultimately requires testing human subjects' actual use of the model in a particular implementation, it is possible to gain some early insights into the question in an implementation-independent way by considering the amount and kind of vocabulary necessary to communicate with the user about types.

We consider the minimum communication necessary to be the generation of type error messages, since if no error messages were ever produced, there would be no benefit to having a type inference system. To be useful, a type error message must indicate three pieces of information:

1. Where in the program the error occurs
2. What the offending type is
3. What the offending type should have been

The second and third pieces of information require a vocabulary of types. Our model's vocabulary of types consists of only three different concepts: primitiveOperations (circleOperations), formIDs (Mirror) and cellIDs (movedPoint). Since programmer-assigned names are used in place of generated IDs, we expect these concepts to be familiar and thus understandable to the programmer. Because of this simple type

vocabulary, we believe our model of types offers a potential for greater understandability than existing type inference models. In particular, the absence of polymorphic types, function types and compositions of types, simplifies the type vocabulary required of the programmer, whereas type inference models that use these concepts must also communicate about them, complicating the type vocabulary.

5.6.3 Permitted types of arguments

Polymorphic references can be viewed as the spreadsheet-based language equivalent of polymorphic function calls. Under this analogy, the RO referenced is the return value and the ROs listed in the defSet list are the arguments. For example, the polymorphic reference illustrated in Figure 5.3 could be interpreted as a function called newPoint which takes arguments oldPoint and x-offset. In most statically typed object-oriented languages, the types of the arguments are constrained to be supertypes or subtypes of some declared type (recall the discussion of covariance and contravariance from Section 2.3.1.1). According to the requirements axiom [R1], however, our model of types only requires that the arguments exist; it says nothing about their types. The reason this omission is acceptable is that type checking will still be performed on the arguments *at edit time* for any new use of this "function." For example, to reuse the function newPoint, the Forms/3 programmer makes a copy of the form Mirror and provides a new formula for cell oldPoint (and possibly for x-offset). Because of the responsiveness of the language, the edit of cell oldPoint's formula causes the formula for cell newPoint to be re-evaluated, generating a new copy of the VADT form appropriate for the value of oldPoint. On this new copy, the "aPoint" and "delta-x" cells have by definition new formulas, referencing cells oldPoint and x-offset respectively, which are statically type checked before they are evaluated. Thus the arguments to a function are always statically type checked at the edit time of the original function call (in the example,

the edit of the argument oldPoint). Since each function call is statically type checked separately from others, there is no reason to impose relationships among the allowed types of the arguments, so the question of contravariance versus covariance of arguments is avoided.

Chapter 6: Future Work

An important next step in the area of similarity inheritance will be empirical work to learn whether users make fewer reuse errors or reuse formulas more often using similarity inheritance as opposed to the current copy/paste/replicate techniques. Two other questions that we would like to explore are whether users are as comfortable with similarity inheritance as with copy/paste/replicate, and whether the explicit representation succeeds at making the flexibility inherent in the approach manageable. Finally, we would like to gather empirical data about whether and how people use mutual inheritance.

Another area for investigation is the issue of priority for name versus similarity. If form Queue, for example, contained both a cell "like top" (but named front) and a cell named top, which one should take priority for polymorphic references to cell top? The current solution is to disallow both name and similarity, that is, the programmer either must not name a cell top or must first break the similarity between Stack's top and Queue's front. We would like to explore whether it is reasonable to make this restriction, or whether both should be allowed, but one given priority. Perhaps a better solution would be to allow both, so that Queue's front can benefit from updates to Stack's top, but give priority to the name top (for polymorphic references) so that the programmer can emphasize interface over implementation.

We would also like to fine-tune the explicit representation devices. For example, the current technique of shading does not distinguish between cells created via large-grained similarity and those created via fine-grained similarity. One solution would be to make the shading for the cells created via fine-grained similarity a little darker than for large-grained similarity. This visual cue would help the programmer see the difference at a glance, rather than having to look at each cell's formula legend. For example, on the

form Deque (Figure 4.2) the abstraction box push and the ROs inside it would appear slightly darker, indicating that they did not come from the form Queue.

Chapter 3 introduced techniques for answering eight of the reuse questions identified in Section 3.2. Future work could investigate ways to help the programmer find answers to the remaining questions. For example, in seeking an answer to the question "what version of this component should I use?" the programmer might benefit from information such as how often a component has been changed, the date of the last change, how often it has been used, and other information regarding the history of the component. Besides additional query attributes such as dates, the repository would benefit from an empirical investigation into how people use it, what they use it for, and in what ways the query interface could be improved.

The current repository focuses on the granularity of entire pre-existing forms, while similarity inheritance supports reuse at a finer granularity on forms as they are currently being programmed. An unresolved research issue is how to extend the repository support to this finer granularity thereby integrating it modelessly into the programming process. The issue of keeping similarity information current in both the repository and the current workspace is complicated by the fact that forms can be affected even when they are not loaded (for example, overriding similarity on the form Queue affects form Stack and how it might appear in the repository even if Stack is not currently loaded).

The type inference system has not yet been implemented. Although we expect most of the translation from axioms to algorithm to be straight-forward, one case deserving special attention is polymorphic references. According to axiom [Gref], a cell picks up its guarantee set from the cell it references. Usually the reference can only change when the cell's formula is edited. In the case of polymorphic references, however, the actual cell referenced (and its type) may change. As discussed in Section

5.6.3, this change also only happens as the result of a program edit, but since the edit is to a cell different from the one containing the polymorphic reference, care should be taken that this case is not missed.

An implementation of a type inference algorithm is also needed to ascertain how significantly the increased space (to store guarantee and requirement sets) and time (to retype-check each use of a polymorphic operation) affect performance, both in terms of overall speed and in terms of maintaining responsiveness. It is possible that even if the speed is much worse than in traditional type checking approaches, it will still be fast enough to be viable for some VPLs, for example, those aimed at end users for writing small programs.

Currently there is a restriction that the type of a cell's value be homogeneous over time. We would like to remove this restriction, investigating the extension of the model to include heterogeneous time, that is, allowing the type of a cell's value to change over time. Removing the restriction would allow formulas that would result in different types at different values of logical time. Such an extension would probably include the intersection of guarantee sets over time. Thus an interesting research question is whether all possible types (over time) can be inferred statically in all cases.

This research was not directly concerned with user interface issues, but because of the visual nature of VPLs it encroaches on that area. A concentration on the user interface aspects of this work would find many areas for design and improvement. For example, the interface for communicating with the user about type errors has not yet been designed. Such an interface could explore the advantages of the live environment of most VPLs to provide the programmer with quick access to the parts of the program involved in the type error. Other user interface features, such as the copy operation **C**, have been designed and implemented as simply as possible. A more sophisticated **C** might include

copying a form directly (by manipulating the form) rather than from the control panel and the ability to copy cells by dragging and dropping from one form to another.

Implementation details: Since Forms/3 is a research prototype, it is undergoing constant development. This is a list of incomplete or missing aspects of the similarity inheritance implementation intended to inform current and future Forms/3 implementors.

- saving and loading similarity information
- cell legend pop-up list (for ancestor lists of more than two)
- similarity arrows in the repository view (needs saving of similarity information)
- inclusion of newly-created forms in the repository (before they are saved)
- propagating cell adds and cuts to copied forms
- generalization: there is a limitation in the current implementation of generalization that restricts the formulas possible in defSets. The defSet cells with "local" formulas do not generalize correctly, so all changes to the copy must use references to off-form cells. (The example in Figure 5.3 did not originally have the cell x-offset on the form Mirror, but instead overrode the 248-Point:delta-x formula as "- (x + x)" which does not generalize properly.)
- an interface for resolving name conflicts when the programmer attempts to paste a cell onto a form that already contains a cell of the same name.

Chapter 7: Conclusion

This dissertation has contributed a model of inheritance called similarity inheritance for declarative, responsive visual programming languages. Adding inheritance to declarative VPLs may improve the ability of the programmer to reuse code, and it is this potential for reuse that we wished to bring not only to programmers but also to end users.

Within this context of reuse, we first explored avenues of support inherently available in declarative VPLs that feature a tight integration between language and environment. The techniques we presented leverage characteristics found in these VPLs to promote code reuse features needed for informal, evolving repositories. Among these features are:

- No special work required of component producers: Our approach supports reuse by a consumer even if the producer has not gone to the work of doing "reusability packaging."
- Saving the consumer from task separation: One of the problems traditionally present in code reuse is that it is often perceived to be easier for consumers to write new code than to find previously-written code, figure out whether it produces the desired results, and learn how to interface with it. By integrating the repository access into the language environment, we are working to address this problem by dissolving the obstacles that separate reusable code from the consumer's working environment.
- Support for informal repositories: Informal repositories do not feature the kinds of controls traditionally followed in "owned" repositories, and hence support cannot be based on the assumption that a particular classification scheme or formalized documentation standard is present. In our approach, components may be only partially-documented or standardized, but they may be explored for reuse using features supported by many VPLs, such as sample values and automatic executions, automatically-derived dataflow views, and the low-level visualizations that are often found in VPLs.

Our repository approach derives the kind of information formerly only available through packaging efforts by code producers and repository administrators, doing so by drawing upon the complete knowledge of language semantics inherent in tightly-integrated VPLs, along with other characteristics found in many such VPLs, such as sample values, liveness, and animation.

From this starting point, we moved on to demonstrate how the code sharing mechanism of similarity inheritance could be integrated into declarative VPLs. We prototyped our approach in the paradigm most widely used by end users: spreadsheets. We have demonstrated how similarity inheritance can be incorporated into the spreadsheet paradigm, using only cells and declarative formulas, without violating the value rule or requiring users to learn other programming languages or macro languages. We have shown also that the approach to inheritance can be used to manage reuse relationships among cells even in simple formula reuse, which traditionally has been supported only by copying or replicating a formula to other cells. This flexibility not only increases the support for this kind of operation, it also affords a gradual path for a user to progress from simple formula copy/paste to more advanced applications of the technique such as inheritance among user-defined types.

An important feature of similarity inheritance is object self-sufficiency. Self-sufficiency enables a concrete style of programming, and avoids the yo-yo problem. These features are a result of the semantic model's reliance on an interaction model. We expect the model to generalize well for incorporation into other kinds of declarative VPLs, the main requirement being that the language be responsive, a characteristic often found in VPLs. Although two of the four elements in the interaction model—the copy operation and the formula definition operation—could be accomplished textually, the other two elements—an explicit representation and a liveness level of 3 or higher—*require* visual elements in the environment that textual languages usually do not

guarantee. Hence, it is unlikely that similarity inheritance would be possible for a textual language.

Approaches to inheritance have traditionally been tied to subtyping, thus allowing (via inclusion polymorphism) objects of subtypes to reuse functions created for supertypes. Since similarity inheritance does not create subtypes, we demonstrated how polymorphic functions can still be attained by introducing a new model of types and an extension of generalization for polymorphism. The model of types for first-order declarative VPLs is suitable for static type inference in the presence of similarity inheritance with operation polymorphism. Static type inference increases the amount of immediate visual feedback that VPLs can potentially provide, because VPLs afford the possibility of immediate (edit-time) feedback as soon as the user introduces a type error. We believe our approach is suitable both for VPLs aimed at programmers and those aimed at end users. Two unique aspects of our model are:

- it separates type requirements from type guarantees, and
- reasoning about each portion of the program is done at the granularity of each operation, rather than at the granularity of entire types.

These aspects have several interesting and desirable effects. The separation of requirements and guarantees allows support even of flexible, fine-grained approaches to inheritance, which supersedes support of traditional, coarse-grained approaches to inheritance. Additionally, this support for fine-grained inheritance is provided without the traditional measure of re-introducing explicit type declarations. Finally, the model allows reasoning to be performed in terms of concrete program units that were explicitly created by the user (such as cells, in Forms/3), rather than in terms of abstract type names, and this is the main reason why the user vocabulary associated with use of this model is small. These properties, while potentially useful in traditional programming languages, are much more critical in declarative VPLs, because their absence prevents the

possibility of using static type inference either in VPLs incorporating similarity inheritance or in VPLs aimed at end users.

Together, the repository techniques and the models of inheritance and types provide powerful language-level code reuse via similarity inheritance for declarative VPLs. The models presented are intended for either programmers or end users and enable a gentle progression toward more sophisticated use as users gain expertise.

Annotated Bibliography

[Abadi and Cardelli 1996] Martin Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401-423, 1996.

Annotation: The original interpretation of matching was in terms of F-bounded subtyping. This paper provides a new interpretation of matching as higher-order subtyping and shows that this interpretation has better properties. (Paraphrased from the introduction: The subtype relation does not hold between some recursively defined object types that arise commonly (that is, in order to preserve soundness they must not be subtypes). A partial solution is to employ 'self' types which works for covariant occurrences of recursion variables. Matching circumvents this problem. Unlike subtyping, matching does not support subsumption, only inheritance.).

[Abadi et al. 1991] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13:237-268, 1991.

Annotation: They use a static type "dynamic" consisting of pairs (value, type) as the type of data that cannot be typed statically such as the result of an arbitrary `eval`.

[Abadi et al. 1993] Martin Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *ACM Symposium on Principles of Programming Languages*, pages 157-167, January 1993.

Annotation: A syntactic approach to studying parametric polymorphism. The authors extend Girard's system F to deal with relations between types.

[Abate et al. 1996] A. F. Abate, M. Nappi, G. Tortora, and M. Tucci. Assisted browsing in a diagnostic image database. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 223-232, Gubbio, Italy, May 1996.

Annotation: Storing and retrieving images based on content. A collection of metadata about an image is used as an iconic index. The example is of medical images for which a physician has identified the items of interest in an image (after automatic edge detection). Images can then be retrieved based on relative locations of items of interest.

[Agesen et al. 1993] Ole Agesen, Jens Palsberg, and Michael Schwartzbach. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. In *European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 247-267. Springer-Verlag, 1993. Also in *Software—Practice and Experience 25:9, pages 975-995, September, 1995.*

Annotation: Self includes both dynamic and multiple inheritance. Type inference is accomplished by deriving and solving type constraints (computing a global fixed-point). A hypothetical program browser is described that makes use of type information.

[Ahlberg and Shneiderman 1994] Christopher Ahlberg and Ben Shneiderman. The alphaslider: A compact and rapid selector. In *Proceedings CHI'94 Human Factors in Computing Systems*, pages 365-371, April 24-28 1994.

Annotation: A usability study tests for speed of locating one movie title from a list of 10,000 using four designs of the alphaslider. They give objective and subjective results and suggest an additional design that takes into account the best features of the designs tested. One application using an alphaslider, the FilmFinder, is illustrated in the colorplates.

[Aiken and Murphy 1991] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *ACM Symposium on Principles of Programming Languages*, pages 279-290, January 1991.

[Aiken et al. 1994] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages*, pages 163-173, Portland, Oregon, January 1994.

Annotation: Type inference systems for dynamically typed languages are called *soft typing*. They present an algorithm for performing type inference for a type language including function types, intersection types, union types, recursive types and conditional types (a kind of control-flow analysis). When a program cannot be proved type-safe, it is not rejected since dynamically typed languages do not impose type constraints, rather explicit run-time type checks are added to the program to make it type safe.

[Alm et al. 1993] Norman Alm, Hohn Todmanm Leona Elder, and A. F. Newell. Computer aided conversation for severely physically impaired non-speaking people. In *Proceedings CHI'93 Human Factors in Computing Systems*, pages 236-241, Amsterdam, the Netherlands, April 24-29 1993.

Annotation: Since conversations follow a general pattern, a user-interface utilizing pre-entered material aimed at use in conversation was constructed and tested. Material is selected according to topic and by selecting me/you, where/what/how/when/who/why and past/present/future. Since only one of these is likely to change at a time, fewer mouse selections are usually needed to find a selection.

[Aloia et al. 1996] N. Aloia, M. Matera, and F. Paternò. A semantics-based approach to designing presentations for multimedia database query results. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 91-100, Gubbio, Italy, May 1996.

Annotation: Describes a method for designing the presentation of a database query result dynamically when the database contains items with multiple or varied representations. They define four composition operations: simple, grouping, merging, and comparison.

[Altmann et al. 1995] Erik M. Altmann and Jill H. Larkin and Bonnie E. John. Display navigation by an expert programmer: A preliminary model of memory. In *CHI Proceedings: Human Factors in Computing Systems*, pages 3-10, Denver, Colorado, May 1995.

Annotation: Proposes a computational cognitive model for the way expert programmers navigate large information displays. Situation knowledge is encoded in long term memory automatically as a by-product of the problem-solving process.

[Amadio and Cardelli 1991] Robert M. Amadio and Luca Cardelli. Subtyping recursive types. In *ACM Symposium on Principles of Programming Languages*, pages 104-118, Orlando, Florida, January 1991.

Annotation: It is difficult to compare recursive types structurally. Simple unfolding is not adequate to decide if two types are equivalent or in a subtype relationship. They show two types are equivalent if they are fixpoints of the same (non-trivial) type context.

[Ambler and Broman 1998] A. Ambler and A. Broman. Formulate solution to the visual programming challenge. *Journal of Visual Languages and Computing*, 9(2):171-209, April 1998.

Annotation: Formulate is a spreadsheet VPL similar in some respects to Forms/3.

[Ambler and Leopold 1998] Allen Ambler and Jennifer Leopold. Public programming in a Web world. In *Proceedings IEEE Symposium on Visual Languages*, pages 100-107, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Uses invisible frames with "submit" protocol rather than sockets for communication between Formulate and a Web front end.

[Atwood et al. 1996] J. W. Atwood, M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang. Steering programs via time travel. In *Proceedings IEEE Symposium on Visual Languages*, pages 4-11, Boulder, Colorado, September 1996.

Annotation: Logical time and steering in Forms/3 with the thermometer and mouse visualization example.

[Baek and Layne 1988] Young K. Baek and Benjamin H. Layne. Color, graphics, and animation in a computer-assisted learning tutorial lesson. *Journal of Computer-Based Instruction*, 15(4):131-135, 1988.

Annotation: In this study, subjects in a computer-assisted learning exercise performed best with animation, and better with graphics than with text only.

[Baker and Eick 1994] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. Technical report, AT&T, 1994.

[Bardou 1996] Daniel Bardou. Delegation as a sharing relationship: Characterization and interpretation. In *ECOOP Workshop on Prototype Based Object Oriented Programming*, 1996.

Annotation: Compares delegation and class-based inheritance on the basis of what types of sharing they allow. Sharing is broken down into name sharing, property sharing and value sharing.

[Baroth and Hartsough 1995] Ed Baroth and Chris Hartsough. Visual programming in the real world. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 2. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

Annotation: Authors found the advantages of visual programming to be a reduction in time to create and modify programs as well as and increased interaction with the customer. The languages used were both dataflow oriented, so the customers, most of whom were familiar with dataflow diagrams, could follow along with the programmer's explanation and understand the program to some extent. The VLs facilitated communication among the developer, customer and computer. Their largest benefit was not for writing code, but rather before the specifications had been determined, the flexibility to work with the customer interactively during the design process, making modifications rapidly to determine the specifications.

[Batory et al. 1994] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirikin. The GenVoca model of software-system generators. *IEEE Software*, September 1994.

Annotation: Talks about Predator, a data structure generator, that scales better than a library of individual data structures and also optimizes well.

[Baumgartner et al. 1996] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, February 1996.

[Bederson et al. 1996a] Benjamin B. Bederson, James D. Hollan, Allison Druin, Jason Stewart, David Rogers, and David Proft. Local tools: An alternative to tool palettes. In *ACM Symposium on User Interface Software and Technology*, pages 169-170, Seattle, Washington, November 1996.

Annotation: Rather than keeping tools in a palette, they are allowed to lie around on the workspace, as on a real desktop. Testing with children has indicated that this is an easier way for them to understand tools. They continue to refine the UI with further user feedback.

[Bederson et al. 1996b] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathan Meyer, David Bacon, and George Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of Visual Languages and Computing*, 7(1):3-31, 1996.

Annotation: Some of the interesting user interface ideas discussed in this article include: different representations of an object depending on the size available to display it (title vs. abstract), representations that depend on the size of the entity displayed (lines of a small file vs. pages of a large file), lenses that change the appearance of entities (column of numbers to bar chart), and visual bookmarks.

[Bhavnani and John 1997] Suresh K. Bhavnani and Bonnie E. John. From sufficient to efficient usage: An analysis of strategic knowledge. In *CHI Proceedings: Human Factors in Computing Systems*, pages 91-98, Atlanta, GA, March 1997.

Annotation: Neither good design nor user experience can guarantee efficient use of computer tools. (An example is resizing several spreadsheet columns to be width X except one in the middle should be width Y. The efficient way is to resize all to X then resize the exception rather than resizing all individually.) This paper discusses strategic knowledge and the particular strategy of aggregation as a powerful tool for efficient operation. Unfortunately, based on empirical study, they find that users tend to have trouble with effective use of aggregation.

[Biddle and Tempero 1995] Robert Biddle and Ewan Tempero. Understanding OOP language support for reusability. In *Workshop on Institutionalizing Software Reuse*, 1995.

Annotation: The authors are interested in the affect of programming language features on code production. They say the advantages of OOP for reuse are (1) design-by-analogy (OOD) leads to using structures that have already been needed and designed before or that will be needed in the future, (2) encapsulation, (3) composition of objects, and (4) making the context code reusable by using inheritance for conformance which allows the context code to work with different subclasses.

[Biddle and Tempero 1996] R. L. Biddle and E. D. Tempero. Understanding the impact of language features on reusability. In *International Conference on Software Reuse*, pages 52-61, Orlando, Florida, April 1996.

Annotation: The authors introduce a conceptual model of reusability concerned with programming language support. They distinguish between reuse of components and context. The model emphasizes dependencies. Checkability: can the language detect if dependencies are met? Customizability: how much control does the context/component have over how it meets the component/context dependencies? Flexibility: does the language introduce dependencies that do not contribute to the behavior of a segment? Intention: are dependencies deliberately introduced by the programmer or because the language requires them or on accident? Language support: can dependencies be described in programming language constructs and can it be automatically checked by the language environment? The authors demonstrate how to apply the model to well-known language features, such as user defined types and classes, that support reusability.

[Biggerstaff and Perlis 1989a] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability: Applications and Experience*, volume 2. Addison-Wesley, Reading, Massachusetts, 1989.

[Biggerstaff and Perlis 1989b] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability: Concepts and Models*, volume 1. Addison-Wesley, Reading, Massachusetts, 1989.

[Biggerstaff and Richter 1987] Ted Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41-49, March 1987.

Annotation: Identifies these reuse problems: conflict between generality and power, conflict between component size (payoff) and reuse potential (less modification), large initial investments. Identifies four problems a reusability system must address: finding components (including similar ones), understanding components (especially to modify them), modifying components, composing components (points out that math composition is straightforward, whole = sum of parts, but both global and local effects make composition more challenging). Problems for code reuse: large numbers of data types, domain not well-understood or constantly changing. Problem for design reuse: representation of design information that is either overly specific or not richly machine processable. A good design representation would represent knowledge about implementation structures in factored form (example, reuse a process table without the accompanying notion of indexed table), allow partial specifications (mixture of precision and fuzziness) that can evolve incrementally, allow flexible couplings between instances and their interpretations (example, process ID and table index), and express controlled degrees of abstraction (implies the first three properties) such that if information can be known based on design so far, it must be representable. Semantic binding (binding by analogy) is offered as a mechanism for breakthroughs in design representation. Their work on Prep is mentioned but no reference given.

[Biggerstaff and Richter 1989] Ted J. Biggerstaff and Charles Richter. Reusability framework, assessment, and directions. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability: Concepts and Models*, volume 1, chapter 1. Addison-Wesley, Reading, Massachusetts, 1989.

Annotation: Identifies reusability dilemmas which take the form of tradeoffs, generality vs. payoff, size (and payoff) vs. reuse potential, and investment vs. benefits (payoff). Identifies four fundamental problems a reusability system must address: finding components, understanding components, modifying components, and composing components. Gives an overview of existing work and research issues, including the reuse of design which they say requires some breakthrough in machine processable representation.

[Biggerstaff 1994] Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Third International Conference on Reuse*, November 1994.

Annotation: Attributes the library scaling problem largely to inadequate abstraction and composition mechanisms in conventional languages. Biggerstaff's solution is to compose at reuse-time and then optimize away the overhead of the levels of abstraction that are no longer necessary at compile-time. Booch's component library is suggested as a benchmark for reuse. Solving the library scaling problem will allow composition of concrete but custom variations generated in a fully hands-off manner (that is, without code glue, component modification, or any other non-black-box reuse technique that eliminates the benefits of reuse). The ideal factorization of features would result in libraries that grow linearly while the set of useful composites grows combinatorily.

[Biggerstaff 1995] Ted J. Biggerstaff. Second order reusable libraries and meta-rules for component generation. In *Workshop on Institutionalizing Software Reuse*, 1995.

Annotation: "First order reuse libraries" (those that are simply reusable building-blocks) by themselves will not (have not) resulted in scalable reuse because composition, refinement and specialization are localized. A "second order reuse library" would allow for components with global effects or transformational rules.

[Blackwell 1996] Alan F. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? In *Proceedings IEEE Symposium on Visual Languages*, pages 240-246, Boulder, Colorado, September 1996.

Annotation: Investigated claims (or perhaps discussion points) made in 140 papers about visual programming languages. Identified 12 themes (such as concreteness and improved productivity), each of which is discussed briefly.

[Blair et al. 1989] Gordon S. Blair, John J. Gallagher, and Javad Malik. Genericity vs Inheritance vs Delegation vs Conformance vs ... *Journal of Object Oriented Programming*, pages 11-17, September 1989.

Annotation: Reexamining Wegner's definition of object-oriented programming. The authors emphasize properties—encapsulation, set-based abstraction, and polymorphism—rather than mechanisms such as classes and inheritance. They separate polymorphism into inclusion polymorphism, when an object belongs to more than one type, and operation polymorphism, when code works with different types of objects. (Wegner did not include operation polymorphism.) Different mechanisms are evaluated according to these properties.

[Blanchet 1995] Bruno Blanchet. Projet d'informatique inférence de types avec dimensions sous caml light. Technical report, INRIA, 1995. In French.

Annotation: Information about the new features of Caml Dim which includes dimension types. Other Caml Light documents are available via ftp at ftp.inria.fr in the directory lang/caml-light.

[Borning et al. 1996] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lecture Notes in Computer Science*, 1106:23+, 1996.

[Borning 1986] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Fall Joint Computer Conference*, pages 36-40, Dallas, Texas, November 1986.

Annotation: Proposes a kind of prototype language providing object classification and updating via constraints maintaining the inheritance relations.

[Bracha and Cook 1990] Gilad Bracha and William Cook. Mixin-based inheritance. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 303-311, October 1990.

Annotation: Examines different mechanisms for using inheritance to implement incremental programming, traditional Smalltalk, Beta's prefixing, CLOS's complex multiple inheritance. Presents a generalized inheritance mechanism with explicit support for mixins that also supports the styles of inheritance in the other three languages. An extension to Modula-3 illustrates their generalized inheritance.

[Braine and Clack 1996] Lee Braine and Chris Clack. Introducing CLOVER: An object-oriented functional language. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop*, number 1268 in Lecture Notes in Computer Science, pages 1-20, 1996.

Annotation: The authors describe CLOVER as 100% functional and 99% object-oriented. Features include referential transparency (no side effects), no multiple inheritance, type-safe (upper bounds on types are known statically) dynamic dispatch based on a single distinguished object, overloading. An interesting aspect is that the distinguished object is the last argument in the list, in order to allow currying. Subclasses must implement (or inherit) every method of their superclass with exactly the same type signatures. The visual interface emphasizes laziness by using a "pull" winder. Methods/functions are stacks of pipes with the result on top.

[Braine and Clack 1997] Lee Braine and Chris Clack. Object-flow. In *Proceedings IEEE Symposium on Visual Languages*, pages 418-419, September 1997.

Annotation: CLOVER is an object-oriented, functional programming language with visual syntax.

[Brooks Jr. 1978] Frederick P. Brooks Jr. *The Mythical Man-Month.* Addison-Wesley Publishing Company, July 1978.

Annotation: A classic software engineering book. By chapters: (1) the difference between a program and a programming project (2) the problem of estimating project time and cost; more workers rarely means less time if you are behind schedule (3) the surgical team approach to programming (4) conceptual integrity—a few good architects should design the interface (5) the second-system effect—frills and embellishments creeping into the design (6) keeping everyone informed of the written specifications (7) communication (keeping a project workbook) and organization (who's the boss?) (8) time and productivity—why only half the work is getting done (9) program size and how to control it (10) documents essential to the manager (11) plan to throw one away; you will anyhow. Plan for change in the program and the

organization. Maintenance is an entropy-increasing process leading to unfixable obsolescence (12) tools, program and document control (13) building programs that work: testing specifications, top down design, structured programming, debugging components, debugging system (14) staying on schedule (15) documentation—combining it with program code.

[Brooks 1997] Kevin M. Brooks. Programming narrative. In *Proceedings IEEE Symposium on Visual Languages*, pages 380-386, September 1997.

Annotation: A tool for authors of fiction, in particular, stories that follow multiple characters and so incorporate multiple points of view.

[Brown and Najork 1996] Marc H. Brown and Marc A. Najork. Collaborative active textbooks: A Web-based algorithm animation system for an electronic classroom. In *Proceedings IEEE Symposium on Visual Languages*, pages 266-275, Boulder, Colorado, September 1996.

[Brown and Vander Zanden 1998] David R. Brown and Brad Vander Zanden. The whiteboard environment: An electronic sketchpad for data structure design and algorithm description. In *Proceedings IEEE Symposium on Visual Languages*, pages 288-295, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Describes tools for the interactive design of data structures and demonstration of algorithms. Designed to be used by teachers of algorithms and data structures.

[Bruce et al. 1993] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1993.

[Bruce et al. 1995] Kim Bruce, Luca Cardelli, Guiseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, pages 221-242, 1995.

Annotation: Covers the issues and diverse views on typing binary methods in an object-oriented language. Explains problems with subtyping and binary methods and privileged access to additional arguments. Evaluates avoiding binary methods, bounded matching, multi-methods and using precise typings as solutions.

[Bruce et al. 1997] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "Match" for object-oriented languages. In *ECOOP Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 104-127. Springer-Verlag, 1997.

Annotation: Describes LOOM, a language using matching that does away with subtyping. Introduces "Hash types" which provide some of the benefits of subtyping. The language is statically-typed (decidably and provably type safe), object-oriented, and provides modules and other information hiding mechanisms.

[Bruce 1995] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1995.

Annotation: Explains problems with current OO typing systems: parameters cannot be changed in subclass overriding methods and return values of methods may return a more general type than we know we have. Introduces MyType as a partial solution.

Matching and bounded matching give the full solution. Bounded matching is comparable to Ada generics and equivalent to the use of F-bounded polymorphism.

[Budd 1991] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

Annotation: A textbook suitable for an introductory course in OOP. Introduces concepts in a language-independent manner, but includes specific comparisons of C++, Smalltalk, Objective-C and Object Pascal.

[Burnett and Ambler 1994] Margaret M. Burnett and Allen L. Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing*, 5(1):29-60, March 1994.

Annotation: Describes the declarative approach to visual abstraction taken by Forms/3. Examines the issues of enforced information hiding, supporting abstraction while preserving concreteness, and defining a type's appearance and behavior.

[Burnett and Gottfried 1998] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, pages 1-33, March 1998.

[Burnett et al. 1994] Margaret Burnett, Richard Hossli, Timothy Pulliam, Brian VanVoorst, and Xiaoyang Yang. Toward visual programming languages for steering in scientific visualization: a taxonomy. *IEEE Computational Science and Engineering*, 1(4):44-62, winter 1994.

[Burnett et al. 1995a] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 28(3):45-54, March 1995.

Annotation: Examines nine major sub-problems of the scaling-up problem for VPLs, static representation, screen real estate, documentation, procedural abstraction, interactive visual data abstraction, type checking, persistence, efficiency and noncoding issues.

[Burnett et al. 1995b] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments.* Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

[Burnett et al. 1998] Margaret M. Burnett, John W. Atwood Jr., and Zachary T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings IEEE Symposium on Visual Languages*, pages 126-133, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Presents an implementation method that supports level 4 liveness in declarative VPLs, ensuring that all values on the screen are correctly updated without unreasonable cost. The method uses lazy evaluation, lazy marking, and culprit tracking. Includes comparisons to other methods.

[Burnett 1991] Margaret M. Burnett. *Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model.* Ph.D. thesis, University of Kansas, August 1991.

[Burnett 1993] Margaret Burnett. Types and type inference in a visual programming language. In *Proceedings IEEE Symposium on Visual Languages*, pages 238-243, Bergen, Norway, August 1993.

Annotation: First paper on the Forms/3 type system.

[Burnett 1995] Margaret M. Burnett. Seven programming languages issues. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 8. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

[Caldiera and Basili 1991] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61-70, February 1991.

Annotation: Describes tools to aid in the process of extracting reusable components ready for the repository from legacy code.

[Caldwell 1994] Bruce Caldwell. Software reuse comes of age. *Information Week*, November 14 1994.

Annotation: Describes a study by QSM Associates of Pittsfield, MA, of 15 software development projects using reuse at nine different companies. They measured 84% reduction in programming costs, 70% reduction in schedule, and lower number of defects. They also note 25-30% more time to produce good reusable code.

[Canning et al. 1989] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object oriented programming. In *Proceedings International Conference on Functional Programming Languages and Computer Architecture*, pages 273-280, September 1989.

Annotation: While bounded quantification allows polymorphism for all subtypes of some type, F-bounded quantification allows polymorphism for all types of a given form which may include references to itself (recursive types).

[Cardelli and Wegner 1985] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471-522, 1985.

[Cardelli 1987] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147-172, 1987.

[Carlson and Burnett 1995] Paul Carlson and Margaret M. Burnett. Integrating algorithm animation into a declarative visual programming language. In *Proceedings IEEE Symposium on Visual Languages*, pages 126-127, Darmstadt, Germany, September 1995.

[Carlson et al. 1996] Paul Carlson, Margaret Burnett, and Jonathan Cadiz. A seamless integration of algorithm animation into a visual programming language. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 194-202, Gubbio, Italy, 1996.

[Carpendale et al. 1997] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. Making distortions comprehensible. In *Proceedings IEEE Symposium on Visual Languages*, pages 36-45, September 1997.

Annotation: Distortions such as fisheye views can disorient and confuse the user in some cases. Guidelines for avoiding this problem are discussed such as the use of grids and shading.

[Cartwright and Fagan 1991] Robert Cartwright and Mike Fagan. Soft typing. In *ACM Conference on Programming Language Design and Implementation*, pages 278-292, Toronto, Ontario, Canada, June 1991.

Annotation: An approach to combining the best of static and dynamic typing. Explicit run-time checks are inserted into code that does not statically type-check. No program is rejected, merely transformed. Includes an algorithm for frugally inserting run-time checks.

[Castagna and Pierce 1994] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *OOPSLA '94*, 1994.

Annotation: Introduction to the limits of F≤ (F-sub).

[Castagna 1995] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431-447, 1995.

Annotation: Contravariance and covariance can and should coexist safely in an object-oriented type system. Contravariance is the correct rule for substitution; covariance is the correct rule for overriding.

[Chalmers et al. 1996] Matthew Chalmers, Robert Ingram, and Christoph Pfranger. Adding imageability features to information displays. In *ACM Symposium on User Interface Software and Technology*, pages 33-39, Seattle, Washington, November 1996.

Annotation: They use a bibliography example to illustrate their landscape view which has colored patches of background to indicate local density. Highlighting of a few randomly chosen objects keeps the display from getting too cluttered (the highlighted objects change so that over time the detail of all objects will have appeared). Titles are also dynamic features. Bias is given to objects closer to the viewer. Queries are as simple as clicking on a word in any visible text. Dynamically updated rings under objects increase in size with the relative search hit frequency. The authors also describe how the view works in a shared environment.

[Chen and Cheung 1997] T. Y. Chen and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Practice*, 9:33-46, 1997.

Annotation: Using program slicing to find bugs. Dicing is the set difference of slices. The idea is to narrow down places where the bug is likely to be by comparing statements executed for different test cases.

[Cherinka et al. 1993] R. Cherinka, C. M. Overstreet, and R. Sparks. Using data flow analysis to determine bi-directional ripple effects during software maintenance. Technical report, Old Dominion University, September 1 1993.

Annotation: Nice introduction and laying out of issues and motivations. Introduction of SCAP, Static Code Analysis Program, as a collection of software engineering tools to examine data flow of imperative languages.

[Citrin et al. 1997] W. Citrin, M. Doherty, and B. Zorn. A graphical semantics for graphical transformation languages. *Journal of Visual Languages and Computing*, 8(2):147-173, April 1997.

[Citrin 1996] Wayne Citrin. Integrating fisheyeing into a visual programming environment. In *Proceedings IEEE Symposium on Visual Languages*, pages 20-27, Boulder, Colorado, September 1996.

Annotation: Discusses the challenges and solutions to incorporating fisheye views into the VIPR environment (a variant of Pictorial Janus). The result preserves containment, performs reshaping, scales both text and graphics, and incorporates zooming.

[Clement et al. 1986] Dominique Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings ACM Conference on Lisp and Functional Programming*, pages 13-27, August 1986.

Annotation: A clear, well-written paper introducing a small type system.

[Cook et al. 1990] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages*, pages 125-135, San Francisco, California, January 1990.

Annotation: They explain why subtypes and subclasses are not the same and present an example functional, explicitly-typed language that allows static type checking.

[Cook 1989] W. R. Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, Nottingham, GB, July 1989.

[Cooper 1995] Alan Cooper. *About Face: The Essentials of User Interface Design*. IDS Books Worldwide, Inc., 1995.

Annotation: Speaking of documents in the most general sense, the author advocates retrieval systems that "remember" information automatically so that the user can later find out if documents have been untouched for a long time, frequently edited, frequently viewed but infrequently edited, etc. (p.107).

[Coplien and Schmidt 1995] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

Annotation: A collection of papers from the first PLoP (Pattern Languages of Programs) conference in August of 1994. These papers have undergone extensive review and editing both before and during the conference which is conducted as a series of writer's workshops.

[Coplien 1991] Jim Coplien. Experience with CRC cards in AT&T. *The C++ Report*, 1991.

[Coplien 1995] James O. Coplien. A generative development-process pattern language. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 13. Addison-Wesley, 1995.

Annotation: A pattern for processes, rather than software design.

[Corridoni et al. 1995] Jacopo Maria Corridoni, Alberto Del Bimbo, and Dario Lucarella. Navigation and visualization of movies content. In *Proceedings IEEE Symposium on Visual Languages*, pages 217-225, Darmstadt, Germany, September 1995.

Annotation: They define a syntax of movies, frames, scenes, and punctuation (dissolves, cuts, etc.) which can be automatically detected. Video information can be searched in a variety of ways providing support for many kinds of users, technical to novice.

[Corridoni et al. 1996] J. M. Corridoni, A. Del Bimbo, S. De Magistris, and E. Vicario. A visual language for color-based painting retrieval. In *Proceedings IEEE Symposium on Visual Languages*, pages 68-75, Boulder, Colorado, September 1996.

[Corridoni et al. 1997] J. Corridoni, A. Del Bimbo, M. Mugnaini, P. Pala, and F. Turco. Pyramidal retrieval by color perceptive regions. In *Proceedings IEEE Symposium on Visual Languages*, pages 205-211, 1997.

Annotation: Color-based searches are handled at different levels of color resolution (from individual pixels to one "average" color for the picture). Only a few levels are needed for good retrieval results.

[Cox and Pietrzykowski 1988] P. T. Cox and T. Pietrzykowski. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. In *Proceedings International Computer Science Conference*, pages 695-704, 1988.

Annotation: Uses the topological sort example to introduce Prograph's dataflow and control concepts. Classes and persistent data are introduced with an and/or game tree example and an employee database.

[Cox and Smedley 1996] Philip T. Cox and Trevor J. Smedley. A visual language for the design of structured graphical objects. In *Proceedings IEEE Symposium on Visual Languages*, pages 296-303, Boulder, Colorado, September 1996.

Annotation: A vision of how extensions to Prograph might enable the "programming of pictures," that is, a generic graphical description of a family of related pictures. Their goal is to combine the power and flexibility of textual design languages with the human comprehension of visual representations.

[Cox and Smedley 1997] Philip T. Cox and Trevor J. Smedley. A declarative language for the design of structures. In *Proceedings IEEE Symposium on Visual Languages*, pages 438-445, September 1997.

Annotation: The language (LSD) is a logic language in which the programmer works more with the objects (that are being designed) than with functions. This is their VL'96 idea with a different and improved approach.

[Cox et al. 1989] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *Proceedings IEEE Workshop on Visual Languages*, pages 150-156, Rome, Italy, October 1989.

Annotation: Introduces Prograph languages. Explains classes, methods, multiplexes and controls using a database browser example. Gives some description of the environment (editor, interpreter and application builder).

[Cox et al. 1995] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 3, pages 45-66. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

Annotation: Reprint of [Cox et al. 1989].

[Cox et al. 1998] Philip T. Cox, Hugh Glaser, and Stuart Maclean. A visual development environment for parallel applications. In *Proceedings IEEE Symposium on Visual Languages*, pages 144-151, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Programmer directives for parallelizing Prograph programs.

[Curtis 1989] Bill Curtis. Cognitive issues in reusing software artifacts. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability: Applications and Experience*, volume 2, chapter 13. Addison-Wesley, Reading, Massachusetts, 1989.

Annotation: Introduces programmer memory and knowledge models. Points out that even "expert" programmers are novices in some domain and thus are less able to organize information about programming in that domain. Programmers tend to move toward uniform ways of organizing domain knowledge as they become more experienced. "...library interface and retrieval aids need to be flexible in order to serve both novice and expert users." Novices need more information to make correct choices among similar solutions, while experts want concise descriptions and fast access.

[Cypher and Smith 1995] A. Cypher and D. Smith. KidSim: End user programming of simulations. In *CHI Proceedings: Human Factors in Computing Systems*, pages 27-34, Denver, Colorado, May 1995.

Annotation: They describe informal user testing on kids and some design changes made. They switched from deep inheritance to simple one-level inheritance. They created jars, groups of objects, that compensate somewhat for the lack of inheritance. They also decided that they needed to allow objects to be larger than one square. They are interested in generalization from concrete examples and they have a notion of 'type' in the generalized rules.

[Czarnecki et al. 1996] Krzystof Czarnecki, Reinhard Hanselmann, Ulrich W. Eisenecker, and Wolfgang Köpf. ClassExpert: A knowledge-based assistant to support reuse by specialization and modification in Smalltalk. In *International Conference on Software Reuse*, pages 188-194, Orlando, Florida, April 1996.

Annotation: ClassExpert is a knowledge-based tool for locating Smalltalk classes according to a functional specification.

[Damas and Milner 1982] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, pages 207-212, Albuquerque, New Mexico, January 1982.

Annotation: They show that their type inference (type assignment) algorithm for purely applicative ML, proved semantically sound elsewhere, finds the most general type possible for every expression and declaration. They also show that it is decidable whether a program is well-typed.

[Dawkes et al. 1996] Huw Dawkes, Lisa A. Tweedie, and Bob Spence. VICKI—the visualisation construction kit. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 257-259, Gubbio, Italy, May 1996.

Annotation: An interesting aspect of this work is the color coding used to indicate degree of mismatch. Rather than hiding all data points that miss, they are shown ranging from black to white depending on the number of criteria for which they fail. The user can see how relaxing one criteria would include more data points, and can do so interactively.

[Djang and Burnett 1998] Rebecca Walpole Djang and Margaret M. Burnett. Similarity inheritance: A new model of inheritance for spreadsheet VPLs. In *Proceedings IEEE Symposium on Visual Languages*, pages 134-141, Halifax, Nova Scotia, Canada, September 1998.

[Djang et al. 1998] Rebecca Walpole Djang, Margaret M. Burnett and Roger D. Chen. Static Type Inference for First-Order Declarative Visual Programming Languages with Inheritance. Submitted to *Journal of Visual Languages and Computing*, July 1998.

[Dony et al. 1992] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 201-217, 1992.

Annotation: Forms/3 doesn't fit their taxonomy because they put message passing as a top-level requirement. If we ignore that, Forms/3 ends up in one of the categories they deem "uninteresting".

[Druin et al. 1997] Allison Druin, Jason Stewart, David Proft, Ben Bederson, and Jim Hollan. Kid pad: A design collaboration between children, technologists, and educators. In *CHI Proceedings: Human Factors in Computing Systems*, pages 463-470, Seattle, Washington, November 1997.

Annotation: Describes iterative design experience and collaboration with children. The goal was a new learning environment incorporating zooming. One of the results of the collaboration is the notion of local tools.

[Duggan and Bent 1996] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37-83, July 1996.

Annotation: also available as University of Waterloo technical report CS-94-14.

[Edwards 1996] Stephen H. Edwards. Representation inheritance: A safe form of "white box" code inheritance. In *International Conference on Software Reuse*, pages 195-204, Orlando, Florida, April 1996.

Annotation: We can safely customize components in a "white box" fashion if the subclasses respect the invariants of the superclass.

[Egenhofer 1996] Max J. Egenhofer. Spatial-query-by-sketch. In *Proceedings IEEE Symposium on Visual Languages*, pages 60-67, Boulder, Colorado, September 1996.

[Eifrig et al. 1995] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Object-Oriented Programming Systems,*

Languages, and Applications (OOPSLA), Austin, Texas, October 1995. Also in ACM SIGPLAN Notices 30(10).

Annotation: Recursively constrained types allow type inference for I-LOOP, a language including let-polymorphism, and multiple inheritance. The distinction between inheritance and subtyping is preserved. Since the types are nonstandard, communicating meaningfully with the programmer about types is a significant problem.

[Eisenstadt 1997] Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30-37, April 1997.

Annotation: Analysis of debugging anecdotes collected by the author from newsgroup-reading programmers. Categorization of why the bug was difficult to track down (top two reasons were Cause/Effect Chasm and Tools Inapplicable or Hampered), how the bugs were found (Data Gathering and "Inspeculation"-inspection, hand simulation, and speculation-were most common), and underlying causes (clobbered memory and vendor problem were at the top). Concludes with suggestions for debugging tools.

[Endres 1993] A. Endres. Lessons learned from an industrial software lab. *IEEE Software*, 10(5):58-61, September 1993.

Annotation: Describes the three factors seen as contributing most to the great increase in quality and productivity experienced at one IBM lab. In a period of 10-12 years, they experienced more than ten times greater quality in shipped code (measured in residual errors after shipment per 1,000 lines of code), doubled productivity and cut project duration roughly in half. The contributing factors named are training in formal methods (although the author advocates semi-formal methods or automation of formal methods), developing for and with reuse, and greater visibility of the organization's process goals (education).

[Erwig and Meyer 1995] Martin Erwig and Bernd Meyer. Heterogeneous visual languages - integrating visual and textual programming. In *Proceedings IEEE Symposium on Visual Languages*, pages 318-325, 1995.

[Evered et al. 1997] M. Evered, J. L. Keedy, A. Schmolitzky, and G. Menger. How well do inheritance mechanisms support inheritance concepts? In *Joint Modular Languages Conferences*, number 1204 in Lecture Notes in Computer Science, pages 252-266, Linz, Austria, March 1997. Springer-Verlag.

Annotation: Defines sixteen concepts that have been realized via inheritance along with lists of requirements for supporting each concept well. Evaluates a few inheritance mechanisms in light of these requirements and offers suggestions for improved inheritance mechanisms.

[Fafchamps 1994] Danielle Fafchamps. Organizational factors and reuse. *IEEE Software*, 11:31, September 1994.

Annotation: Examines four models of producer-consumer-manager organizational models. For Hewlett-Packard, the team producer model (with separate manager) worked best.

[Flanagan and Felleisen 1997] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *ACM Conference on Programming Language Design and Implementation*, 1997.

Annotation: This paper provides an improvement over simple set-based analysis. The new algorithm can handle larger programs.

[Flanagan et al. 1996] Cormac Flanagan, Matthew Flatt, Shiram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the Web of program invariants. In *ACM Conference on Programming Language Design and Implementation*, pages 23-32, PA, USA, May 1996.

Annotation: This paper is difficult to classify because it is about debugging, yet it is static program analysis of sets of values, i.e. types. Sets of possible values are computed for each expression in the program. A (data) flow graph models how values flow through the program and errors typically caught only at run-time can be detected, such as subscript out of bounds, division by zero, and dereferencing nil pointers. MrSpidey includes a user interface for this analysis which displays value sets, arrows to value sources, etc. for the Scheme language.

[Frakes and Terry 1995] William Frakes and Carol Terry. Software reuse and reusability metrics and models. Technical Report TR-95-07, Virginia Polytechnic Institute and State University, 1995.

[Frakes et al. 1995] W. Frakes, R. Prieto-Díaz, and C. Fox. DARE: Domain analysis and reuse environment. In *Workshop on Institutionalizing Software Reuse*, St. Charles, Illinois, 1995.

[Freeman et al. 1996] Elisabeth Freeman, David Gelernter, and Suresh Jagannathan. Uniformity of environment and computation in MAP. In *Proceedings IEEE Symposium on Visual Languages*, pages 130-137, Boulder, Colorado, September 1996.

Annotation: Describes how the MAP language's meta-commands allow visual program characteristics (such as color and transparency) to be changed programatically.

[Friedman et al. 1994] D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages*. MIT Press/McGraw-Hill, second edition, 1994.

Annotation: Programming Languages textbook using small interpreters built in scheme to teach the basics. Includes some more advanced topics as well, such as type inference in the new chapter for the 2nd Edition.

[Gamma et al. 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

Annotation: A collection of 23 design patterns for object-oriented programming with an emphasis on C++ and Smalltalk. The catalog is composed of patterns that help with creating, structuring, and organizing the behavior of classes and objects. The introduction describes what a pattern is and how to select and use an appropriate pattern. A case study of the design of a document editor illustrates the use of eight patterns.

[Gannon and Horning 1975] John D. Gannon and J. J. Horning. Language design for reliability. *IEEE Transactions on Software Engineering*, 1(2), June 1975.

Annotation: Discusses some language features (for imperative languages) that can reduce the frequency and/or persistence of errors.

[Garlan et al. 1995] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, pages 17-26, November 1995.

Annotation: Even if the programming language, platforms, and database schemas are compatible, software components and connectors are difficult to reuse together because of architectural mismatch—incompatible assumptions about the system structure. These assumptions include: where the control lies (can two event loops be merged?), how component data is manipulated, what infrastructure is necessary, the kinds of data that can be communicated, and the topology of system communications. Suggested forward directions: make the architectural assumptions explicit, create components whose architectural assumptions are not spread throughout and can be modified by module substitution, create bridging technologies that mediate between components or negotiate interfaces, codify and disseminate principles and rules for software composition.

[Garzotto et al. 1996] Franca Garzotto, Luca Mainetti, and Paolo Paolini. Modal navigation for hypermedia applications. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 59-66, Gubbio, Italy, May 1996.

Annotation: Argues for different levels of hypermedia interaction for different users. They identify six modes that can vary independent of the others: type of media, rhetorical style, language, fruition (degree of active involvement), length, and topology (sequence, lattice, etc.). The Polyptych system, a museum presentation for casual visitors, intentional visitors, and experts, is described. They also point out that decisions on modality can be made at installation, initially (and at any further time) by the user, by some default that is modifiable by the user, or by automatic adaptation of the system.

[Gilmore et al. 1995] David J. Gilmore, Karen Pheasey, Jean Underwood, and Geoffrey Underwood. Learning graphical programming: An evaluation of KidSim(tm). In *INTERACT: Proceedings of the IFIP Conference on Human-Computer Interaction.*, 1995.

Annotation: Children using KidSim did not learn abstraction or other programming skills. One of the problems was the ability to overwrite rules rather than edit them; if something didn't work right, it was rewritten instead of edited.

[Gilmore 1995] David J. Gilmore. Interface design: Have we got it wrong? In *INTERACT: Proceedings of the IFIP Conference on Human-Computer Interaction.*, 1995.

Annotation: Gilmore suggests that we examine the goals of our interfaces more carefully. His experiment suggests that speed and learning are opposing goals and that "better" may not have the same meaning for all interfaces. That is, when we make the interface very fast and easy to experiment with, users perform faster but do not learn the concepts related to success.

[Gindling et al. 1995] Jim Gindling, Andri Ioannidou, Jennifer Loh, Olav Lokkebo, and Alexander Repenning. LEGOsheets: A rule-based programming, simulation and manipulation environment for the LEGO programmable brick. In *Proceedings IEEE Symposium on Visual Languages*, pages 172-179, 1995.

[Glaser and Smedley 1995] Hugh Glaser and Trevor J. Smedley. P sh-the next generation of command line interfaces. In *Proceedings IEEE Symposium on Visual Languages*, pages 29-36, Darmstadt, Germany, September 1995.

Annotation: They compare the development of scripting and shell needs on Macintoshes with the development of Pidgin languages. They propose an interpreted Prograph-like visual shell for creating, modifying and executing shell commands.

[Gold and Rosson 1991] Eric Gold and Mary Beth Rosson. Portia: An instance-centered environment for Smalltalk. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 62-74, 1991.

Annotation: A development environment that strives to mirror the programmer's style of thinking by allowing the programmer to tinker with real instances in the environment.

[Goldberg and Richardson 1993] David Goldberg and Cate Richardson. Touch-typing with a stylus. In *Proceedings of CHI'93 Human Factors in Computing Systems*, pages 80-87, Amsterdam, The Netherlands, April 24-29 1993.

Annotation: Most handwriting recognition systems have only novice mode corresponding to hunt-and-peck on a keyboard. Using a modified alphabet of one-stroke letters can result in nearly 3 letters per second entry rate (touch typists can achieve 6-7). Letters can be written on top of each other since the end of a stroke = end of letter. They conjecture that further speedup would result from combining Speedwriting with unistrokes.

[Goldberg and Robson 1983] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gorlick and Quilici 1994] Michael Gorlick and Alex Quilici. Visual programming-in-the-large versus visual programming-in-the-small. In *Proceedings IEEE Symposium on Visual Languages*, pages 137-144, St. Louis, Missouri, October 1994.

Annotation: Describes a visual software engineering environment and how it addresses some of the scaling problems for visual programming-in-the-large. Uses zooming to handle documentation and annotation. Uses partial construction (types of inputs and outputs) to aid component discovery.

[Gottfried and Burnett 1997] H. Gottfried and M. Burnett. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. In *Proceedings IEEE Symposium on Visual Languages*, pages 246-253, February 1997. Also Oregon State University TR 97-60-2.

Annotation: Adding gestural specification of types (even user-defined types) to the spreadsheet paradigm.

[Graham et al. 1996] T. C. Nicholas Graham, Catherine A. Morton, and Tore Urnes. Clockworks: Visual programming of component-based software architectures. *Journal of Visual Languages and Computing*, 7:175-196, 1996.

Annotation: Pieces of the architecture can be grouped; groups can be arbitrarily nested. They claim that the flexibility encourages programmers to maintain reasonable structure and "create elegant group definitions, aiding the development of reusable component libraries". They also say that the key to scalability is good support for hiding information and that designing a visual language is more closely related to designing a user interface.

[Green and Petre 1996] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, pages 131-174, 1996.

Annotation: Cognitive dimensions are terms describing the cognitively important aspects of a programming language, such as "closeness of mapping to the domain" and "hidden dependencies.".

[Griss and Wentzel 1995] Martin L. Griss and Kevin D. Wentzel. Hybrid domain-specific kits. *Journal of Systems Software*, 30:213-230, 1995.

Annotation: Defines hybrid kits for systematic, domain-specific software reuse and discusses how to create and use them. Presents a kit evaluation framework and sample evaluations of Visual Basic and their own to-do-list management kit. Notes the importance of openness (the ability to add components) to a reuse kit.

[Griss and Wosser 1995] Martin Griss and Marty Wosser. Making reuse work at Hewlett-Packard. *IEEE Transactions on Software Engineering*, 12(1):105-107, January 1995.

[Griss 1995] Martin L. Griss. Software reuse: Objects and frameworks are not enough. *Object Magazine*, pages 77-79,87, February 1995.

[Grundy and Hosking 1995] John C. Grundy and John G. Hosking. ViTABaL: a visual language supporting design by tool abstraction. In *Proceedings IEEE Symposium on Visual Languages*, pages 53-60, Darmstadt, Germany, September 1995.

Annotation: Vitabal is a visual language and environment supporting the design and implementation of software using the tool abstraction paradigm which supports functional evolution.

[Grundy et al. 1995] John Grundy, John Hosking, Stephen Fenwick, and Warwick Mugridge. Connecting the pieces. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 11. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

Annotation: Describes a visual software engineering environment that supports both visual and textual representation, allowing the programmer to choose which to use for a given task. Different views are automatically kept consistent.

[Hall 1993] Robert J. Hall. Generalized behavior-based retrieval. In *International Conference on Software Engineering*, pages 371-380. ACM Press, 1993.

Annotation: Based on Behavior Sampling, proposed by Podgurski and Pierce, the GBR prototype searches for components and small combinations of components that return the desired output given sample input-output pair(s). Functional models are used when components are not side-effect free, and constants or optional inputs allow the user to supply only the most important arguments as input.

[Harris 1991] Warren Harris. Contravariance for the rest of us. *Journal of Object-Oriented Programming*, 4(7):10-18, November/December 1991.

Annotation: Describes the notion of contravariance and why it is important to object-oriented programming. Gives five ways that C++ programmers circumvent the problem of subclasses not being subtypes and overloading not performing what is actually desired.

[Harrison 1995] Susan M. Harrison. A comparison of still, animated, or nonillustrated on-line help with written or spoken instructions in a graphical user interface. In *CHI*

Proceedings: Human Factors in Computing Systems, pages 82-89, Denver, Colorado, May 1995.

Annotation: "Results consistently revealed that visuals, either still graphic or animated, in the on-line help instructions enabled the users to significantly perform more tasks in less time and with fewer errors" than users with only textual help. Animation alone did not appear to benefit participants in this study.

[Hearst 1995] Marti A. Hearst. Tilebars: Visualization of term distribution information in full text information access. In *CHI Proceedings: Human Factors in Computing Systems*, 1995.

Annotation: Tilebars indicate at a glance the distribution and density of occurrences of keywords in a text file.

[Hendry 1995] David G. Hendry. Display-based problems in spreadsheets: a critical incident and a design remedy. In *Proceedings IEEE Symposium on Visual Languages*, pages 284-290, Darmstadt, Germany, September 1995.

Annotation: Users of spreadsheets don't always want relative vs. absolute, sometimes a relative-with-offset is desired. The system can easily compute this kind of offset given two examples and the user doesn't have to bother with dollar sign syntax any more.

[Henninger 1994] Scott Henninger. Using iterative refinement to find reusable software. *IEEE Software*, pages 48-59, September 1994.

Annotation: People often lack a clear idea of what they are searching for when they construct a query. CodeFinder allows incrementally constructed queries and supports retrieval of related items. Example retrieved items facilitate quick modifications to the query by direct manipulation of the description. Thesaurus construction is automatic.

[Hild and Poulovassilis 1996] Stefan G. Hild and Alexandra Poulovassilis. Hyperlog: a system for database querying and browsing. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 260-262, Gubbio, Italy, May 1996.

Annotation: The interesting thing about Hyperlog is that it uses the same representation for data, query and results.

[Hirakawa et al. 1998] Masahito Hirakawa, Priyantha Hewagamage, and Tadao Ichikawa. Situation-dependent browser to explore the information space. In *Proceedings IEEE Symposium on Visual Languages*, pages 108-115, Halifax, Nova Scotia, Canada, September 1998.

Annotation: They explore the use of a new framework for personal information management (as opposed to the desktop metaphor) called Situation Information Filing and Filtering (SIFF). The browser organizes information based on time, place and kind of document/activity and allows searching and filtering based on these same attributes.

[Hoadley et al. 1996] Christopher M. Hoadley, Marcia C. Linn, Lydia M. Mann, and Michael J. Clancy. When, why and how do novice programmers reuse code? In *Empirical Studies of Programmers*, Norwood, NJ, 1996.

Annotation: Both negative attitudes toward reuse and poor program comprehension contribute to students not reusing code. 20% of the studied students thought they shouldn't reuse code; reasons ranged from viewing reuse as tedious, deficient, or plagiarism. Students with an abstract (top-level rather than algorithmic)

understanding were more likely to reuse code. They conclude that CS classes should emphasize the benefits of reuse and foster abstract understanding of code.

[Hooper and Chester 1991] James W. Hooper and Rowena O. Chester. *Software Reuse: Guidelines and Methods*. Software Science and Engineering. Plenum Press, NY, 1991.

Annotation: A summary of current research in software reuse and practical suggestions for integrating reuse into the software engineering process.

[Hudson et al. 1997] Scott E. Hudson, Roy Rodenstein, and Ian Smith. Debugging lenses: A new class of transparent tools for user interface debugging. In *ACM Symposium on User Interface Software and Technology*, pages 179-187, October 1997.

Annotation: Debugging lenses make visible the parts of user interfaces that normally must be explored textually such as location attributes, bounding boxes, and constraint relationships. Their example lenses have tools along the sides that affect the display.

[Hudson 1994] Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3):209-239, July 1994.

Annotation: A language based on the spreadsheet paradigm for specifying user interfaces. Cells are grouped into objects which can inherit from other objects (each object has a "like" cell naming the parent, if any). Inheritance is on the level of an entire object with overriding of individual cells. Penguims has a procedure for handling object deletion in the presence of inheritance. Features that violate the spreadsheet paradigm include imperative code and interactors.

[Ibrahim 1998a] Bertrand Ibrahim. Diagrammatic representation of data types and data manipulations in a combined data- and control-flow language. In *Proceedings IEEE Symposium on Visual Languages*, pages 262-269, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Abstract visual representations of data types and their manipulations.

[Ibrahim 1998b] Bertrand Ibrahim. Optimizing cut-and-paste on directed graphs, with a user-controlled edge reconstruction strategy. In *Proceedings IEEE Symposium on Visual Languages*, pages 90-91, Halifax, Nova Scotia, Canada, September 1998.

Annotation: (poster) A mechanism for graph editing that has the potential to save the user a few steps in cleaning up edge connections. The default selection consists of all nodes within the user's rubberband selection area plus edges such that entry and exit points are isolated (the strategy is to try to have one entry point and one exit point for the selection). The user may of course adjust the selection before cutting. Pasting can be performed "in space" or at a graph point; in the latter case, the selection's entry and exit points are matched to the paste point when possible.

[IEEE Software 1996] Special issue on systematic reuse. *IEEE Software*, September 1996.

[Igarashi et al. 1998] Takeo Igarashi, Jock D. Mackinlay, Bay-Wei Chang, and Polle T. Zellweger. Fluid visualization of spreadsheet structures. In *Proceedings IEEE Symposium on Visual Languages*, pages 118-125, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Techniques for making spreadsheet dataflow visible. Includes static and dynamic views as well as navigation techniques. Some neat stuff! Also a nice video at the conference.

[Ingalls et al. 1988] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, and K. Doyle. Fabrik, a visual programming environment. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 76-190, San Diego, California, September 1988.

[Jamal and Wenzel 1995] Rahman Jamal and Lothar Wenzel. The applicability of the visual programming language LabVIEW to larger real-world applications. In *Proceedings IEEE Symposium on Visual Languages*, pages 99-106, Darmstadt, Germany, September 1995.

Annotation: They present a case study of ultrasonic scanners, commenting on debugging, documentation, rapid prototyping and experimental programming. They emphasize the ease of use for the engineers and the speed of development and change made possible by the key concepts of virtual instrument abstraction, hierarchy and integrated user interface.

[Jerding and Stasko 1995] Dean F. Jerding and John T. Stasko. The information mural: a technique for displaying and navigating large information spaces. In *Proceedings IEEE Visualization Symposium on Information Visualization*, pages 43-50, Atlanta, GA, October 1995.

[Johnson and Shneiderman 1991] Brian Johnson and Ben Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the International IEEE Visualization Conference*, pages 284-291, San Diego, California, 1991.

[Jones 1995] Mark P. Jones. Simplifying and improving qualified types. In *ACM Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995.

Annotation: An improved type inference algorithm for qualified types.

[Jones 1997] Mark P. Jones. First-class polymorphism with type inference. In *ACM Symposium on Principles of Programming Languages*, January 1997.

Annotation: In the Hindley-Milner type system, polymorphic values are not first-class. The type system FCP supports first-class polymorphism without giving up type inference.

[Joos 1994] Rebecca L. Joos. Software reuse at Motorola. *IEEE Software*, 11:42-47, September 1994.

Annotation: Experience with introducing reuse into the software development process at Motorola. Describes what happened pointing out what they did right and wrong at each step in hindsight. Two pilot programs are also described. The first is a cash reward incentive program that has more than paid for itself.

[Jouvelot and Gifford 1991] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303-310, Orlando, Florida, January 1991.

Annotation: (By reconstruction, they mean inference.) They present a type inference algorithm for a polymorphic typed language with types and effects (effects describe how expression compute-they mention store, communication and control effects) and with first class procedures. Effects seem to generally be useful for scheduling concurrency safely. What caught my attention is the use of constraints in their type system and the fact that they say record types are the closest related work in algebraic reconstruction.

[Jun and Michaelson 1998] Y. Jun and G. Michaelson. A visualization of polymorphic type checking. *Journal of Functional Programming*, 1998. to appear.

Annotation: Describes the use of colors to visualize types in a functional language.

[Kamba et al. 1996] Tomonari Kamba, Shawn A. Elson, Terry Harpold, Tim Stamper, and Piyawadee "Noi" Sukaviriya. Using small screen space more efficiently. In *CHI Proceedings: Human Factors in Computing Systems*, pages 383-390, 1996.

Annotation: Transparent icons. Using icons at 20% visibility and overlapping hypertext at 80% visibility along with a delay for selecting the secondary item, this study found that users initially made mistakes selecting, but learned quickly. Users overwhelmingly preferred that the icons be the secondary item and wanted the shorter delay time. They also expressed a desire for more immediate response when selecting icons that did not overlap selectable text.

[Karsenty 1996] Laurent Karsenty. An empirical evaluation of design rationale documents. In *CHI Proceedings: Human Factors in Computing Systems*, pages 150-156, 1996.

Annotation: The conclusions of this study are: designers attempting to understand a new design do ask design rationale (DR) questions, some designers use DR documents opportunistically while others read the entire document to understand each issue, 41% of designers questions were answered by the DR document.

[Katzenelson and Pinter 1992] Jacob Katzenelson and Shlomit S. Pinter. Type matching, type graphs, and the Schanuel conjecture. *ACM Transactions on Programming Languages and Systems*, 14(4):574-588, 1992.

[Kay 1984] Alan Kay. Computer software. *Scientific American*, pages 53-59, September 1984.

Annotation: Includes Kay's value rule for spreadsheets which states that a cell's value is defined solely by its formula.

[Kennedy 1994] Andrew Kennedy. Dimension types. In *European Symposium on Programming*, number 788 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

Annotation: Extends a strongly typed programming language with the notion of dimension types (seconds, kilometers, etc.). Discusses some problems with the system and sketches some more powerful systems.

[Kimura et al. 1990] T. D. Kimura, J. W. Choi, and J. M. Mack. Show and tell. In E. P. Glinert, editor, *Visual Computing Environments*. IEEE Computer Society Press, Washington, DC, 1990.

[Kiper et al. 1997] James D. Kiper, Elizabeth Howard, and Chuck Ames. Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing*, 8:175-192, 1997.

Annotation: Identifies five criteria with metrics for evaluating and comparing VPLs. The criteria are visual nature, functionality, ease of comprehension, paradigm support and scalability.

[Koike and Chu 1997] Hideki Koike and Hui-Chu Chu. VRCS: Integrating version control and module management using interactive three-dimensional graphics. In *Proceedings IEEE Symposium on Visual Languages*, pages 168-173, September 1997.

Annotation: A visual interface and visualization system for RCS version control.

[Koike et al. 1996] Yuichi Koike, Yasuyuki Maeda, and Yoshiyuki Koseki. Enhancing iconic program reusability with object sharing. In *Proceedings IEEE Symposium on Visual Languages*, pages 288-295, Boulder, Colorado, September 1996.

Annotation: Describes two mechanisms for enabling reuse of objects in their language: customization (which appears to be extension) and combination. They require the ability to propagate changes and flexibility since the designer cannot anticipate all future needs. Their method is specifically for node-and-wire languages, what they call iconic languages. Claimed advantages are: propagation of changes, flexibility, ease of use, and supporting inheritance. Examples are from the GUI domain. The editor for sharing objects displays only valid choices: shareable objects.

[Koltun and Hudson 1991] Philip Koltun and Anita Hudson. A reuse maturity model. In *Workshop on Institutionalizing Software Reuse*, Reston, Virginia, November 1991.

[Korson and McGregor 1990] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40-60, September 1990.

Annotation: Basic concepts, including two roles of inheritance: "is-a" and reuse.

[Korson 1996] Tim Korson. Managing reuse: Applying the law of gravity. *Object Magazine*, 6(2):34-36, April 1996.

Annotation: Lists four axioms of reuse: (1) reuse limited to class libraries will not fundamentally increase productivity (2) a component is not ready for general reuse until it has been used three times (3) the likelihood that a developer will reuse a component decreases greatly as the distance to it increases (4) an organization should not try to attain the maximum amount of reuse. The author advocates a *corporate reuse portfolio* consisting of a set of reuse assets at various levels of abstraction.

[Kristensen and Osterbye 1996] Bent Bruun Kristensen and Kasper Osterbye. A conceptual perspective on the comparison of object-oriented programming languages. *ACM SIGPLAN Notices*, 31(2):42-54, February 1996.

Annotation: Uses concepts rather than language features to compare OO languages.

[Krohn 1996] Uwe Krohn. VINETA: Navigation through virtual information spaces. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 49-58, Gubbio, Italy, May 1996.

Annotation: They use *flying arrows* to represent keywords in three-dimensional space. Documents are spheres with markers, color, and dimness indicating a

document's contents with respect to the keywords selected. Locations of documents and keyword arrows are determined by a biplot of a document-keyword matrix. Most of the paper is devoted to an explanation of the math involved.

[Kurlander 1993] D. Kurlander. Chimera: Example-based graphical editing. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, chapter 12. MIT Press, 1993.

Annotation: Uses a comic-strip style graphical history to record user actions. The histories are then editable and can also be made into reusable macros (procedures).

[Landauer and Hirakawa 1995] Jurgen Landauer and Masahito Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proceedings IEEE Symposium on Visual Languages*, pages 267-274, Darmstadt, Germany, September 1995.

Annotation: PBD has problems with control structure and generalization. They propose visual awk which uses the simple do-for-every control structure. Generalization uses vertical demonstration or as many examples as possible and takes advantage of human visual scanning abilities. They use a spreadsheet metaphor for a text editing prototype.

[Läufer and Odersky 1994] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411-1430, September 1994.

Annotation: They treat abstract types as first class.

[Läufer 1996] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, June 1996.

Annotation: Introduces a language with type classes and existential types and develops a type system and inference algorithm for it. Also includes a formal semantics using 2nd order lambda-calculus and shows soundness of the type system.

[Launchbury 1991] John Launchbury. *A Strongly-Typed Self-Applicable Partial Evaluator*, volume 523 of *Lecture Notes in Computer Science*, pages 145-164. Springer-Verlag, Cambridge, MA, USA, 1991.

Annotation: A partial evaluator written in and for a strongly-typed language.

[Launchbury 1993] John Launchbury. A natural semantics for lazy evaluation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144-154, January 1993.

Annotation: The author defines an operational semantics for lazy evaluation which provides an accurate model for sharing. The model is suitable for studying space behavior of terms under lazy evaluation.

[Lawrence et al. 1994] A. W. Lawrence, A. M. Badre, and J. T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings IEEE Symposium on Visual Languages*, pages 48-54, St. Louis, Missouri, October 1994.

Annotation: This paper describes a user study examining the effects of algorithm animation on learning. Their results indicate that active involvement in the animation (in the study this consisted of providing a graph for the Kruskal MST algorithm) did significantly improve students' understanding.

[Leopold and Ambler 1997] Jennifer Leopold and Allen Ambler. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings IEEE Symposium on Visual Languages*, pages 28-35, September 1997.

Annotation: Explores using the combination of voice, handwriting and gestures as input mechanisms for Formulate formulas and for some navigation and object manipulation.

[Leroy and Mauny 1991] Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Proceedings of the 5th ACM conference*, volume 523 of *Lecture Notes in Computer Science*, pages 406-426. Springer-Verlag, 1991.

Annotation: Using `dyn` type to integrate required run-time type checks (needed for `eval` and `intern`) into a statically-typed language.

[Leroy and Weis 1991] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *ACM Symposium on Principles of Programming Languages*, pages 291-302, Orlando, Florida, January 1991.

Annotation: integrating imperative programming style with applicative kernel of ML, allows generic functions over mutable data structures to have fully polymorphic types by restricting type generalization-they prohibit the use of a reference with two different types.

[Lewis 1990] Clayton Lewis. NoPumpG: Creating interactive graphics with spreadsheet machinery. In Ephraim P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 526-546. IEEE Computer Society Press, Los Alamitos, California, 1990.

[Li et al. 1997] Xiaosong Li, Warwick Mugridge, and John Hosking. A petri net-based visual language for specifying GUIs. In *Proceedings IEEE Symposium on Visual Languages*, September 1997.

Annotation: Introduces a mechanism for procedural abstraction for Petri nets, through a generative technique reminiscent of Forms/3's constr-names.

[Lieberman and Fry 1995] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *CHI Proceedings: Human Factors in Computing Systems*, pages 480-486, Denver, Colorado, May 1995.

Annotation: They introduce ZStep 94, a program debugging environment designed to help the programmer understand the correspondence between static code and dynamic execution. The programmer can step backward as well as forward through the program, either by code fragment or according to the next (previous) graphical step. Also provides exploration of previous program values.

[Lieberman 1986a] Henry Lieberman. Concurrent object-oriented programming in Act 1. In Yonezawa and Tokoro, editors, *Concurrent Object-Oriented Programming*, pages 9-36. MIT Press, Cambridge, MA, 1986.

Annotation: Actors as objects: classic prototypes with delegation along with lots about concurrency.

[Lieberman 1986b] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Object-Oriented Programming Systems,*

Languages, and Applications (OOPSLA), pages 214-223, 1986. Also in SIGPLAN Notices 21(9).

[Lieberman 1995] Henry Lieberman. The visual language of experts in graphical design. In *Proceedings IEEE Symposium on Visual Languages*, pages 5-12, Darmstadt, Germany, September 1995.

Annotation: Graphic Design knowledge is passed on by means of visual examples. Text cannot adequately represent the information. Students generalize from the examples and make analogies from them. Conditionals and abstraction in this visual realm are achieved through multiple examples and representative sketches respectively.

[Lim 1996a] Wayne C. Lim. Legal and contractual issues in software reuse. In *International Conference on Software Reuse*, Orlando, Florida, April 1996.

Annotation: Considers the benefits and drawbacks of copyright, trade secrets, and patents for property protection. Discusses issues that should be covered in contracts governing software reuse.

[Lim 1996b] Wayne C. Lim. Reuse economics: A comparison of seventeen models and directions for future research. In *International Conference on Software Reuse*, pages 41-50, Orlando, Florida, April 1996.

Annotation: Discusses the similarities and differences among seventeen different reuse economic models. Identifies the most common inputs and outputs of the models and suggests areas for future research.

[Liskov and Wing 1994] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841, November 1994.

Annotation: A definition of subtype, and two formal methods for proving that types do not violate the defined properties. They require a subtype to be transparently substitutable for any of its supertypes even in the presence of aliasing and concurrent data sharing.

[Ludolph et al. 1988] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. The Fabrik programming environment. In *Proceedings IEEE Workshop on Visual Languages*, pages 222-230, Pittsburgh, Pennsylvania, October 1988.

[Mackay 1995] Wendy E. Mackay. Ethics, lies and videotape... In *Proceedings of CHI'95 Human Factors in Computing Systems*, pages 138-145, Denver, Colorado, May 7-11 1995.

Annotation: Sets forth guidelines for ethical use of video. Gives several examples of potentially embarrassing or slanderous use of video.

[Malefant 1996] Jacques Malefant. Object-centered programming. In *ECOOP Workshop on Prototype Based Object Oriented Programming*, 1996.

Annotation: In this position paper the author argues "concreteness need not be the enemy of abstraction." Instead of rejecting abstraction in the form of classes and then letting it in the back door in the form of traits and maps (as in Self), language designers should realize that the focus of prototype-based languages is that the programming experience be object-centered. Abstractions must be introduced at some

stage in the development process, why not allow development to progress from concrete objects to generalizations in the later stages?

[Malenfant et al. 1996] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A semantics of introspection in a reflective prototype-based language. *Lisp and Symbolic Computation*, 9(2/3):153-180, May/June 1996.

[Marriott and Meyer 1997] Kim Marriott and Bernd Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8, 1997.

Annotation: Presents a grammar hierarchy for VPLs similar to the Chomsky Hierarchy. Since all VPLs are context sensitive in the Chomsky sense, they redefine what context sensitive means for VPLs.

[Martin and Hankin 1987] Chris Martin and Chris Hankin. *Finding Fixed Points in Finite Lattices*, pages 426+. Lecture Notes in Computer Science. Springer-Verlag, 1987.

Annotation: abstract interpretation of declarative languages, strictness analysis, finding fixpoints of recursive functions.

[McWhirter 1996] Jeffrey D. McWhirter. AlgorithmExplorer: A student-centered algorithm animation system. In *Proceedings IEEE Symposium on Visual Languages*, pages 174-181, Boulder, Colorado, September 1996.

Annotation: A student-centered approach to animation allowing 3 tiers of sophistication and complexity. The system is available for outside use.

[Meuter et al. 1996] Wolfgang De Meuter, Tom Mens, and Patrick Steyaert. Agora: Reintroducing safety in prototype-based languages. In *ECOOP Workshop on Prototype Based Object Oriented Programming*, 1996.

Annotation: Addresses two safety issues most prototype-based languages have. The encapsulation problem exists in languages that provide dynamic object extension. The prototype corruption problem is when a prototype is modified accidentally instead of a clone. The authors propose encapsulated versions of extension and cloning that do not allow these problems.

[Meyer 1992] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Meyer 1998] Bernd Meyer. Competitive learning of network diagram layout. In *Proceedings IEEE Symposium on Visual Languages*, pages 56-63, Halifax, Nova Scotia, Canada, September 1998.

Annotation: A fast and pretty good graph layout algorithm based on a competitive learning algorithm (which extends self-organization strategies from neural networks). A very nice demonstration applet is available via
`http://www.bounce.to/BerndMeyer`.

[Mili et al. 1995] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528-561, June 1995.

Annotation: A discussion of reuse research including managerial aspects but concentrating on the technical aspects. Covers process, measurement, acquiring

components, object-oriented analysis and design, object-oriented programming, component retrieval, component composition and adapting components.

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

[Minas and Shklar 1996] Mark Minas and Leon Shklar. A high-level visual language for generating web structures. In *Proceedings IEEE Symposium on Visual Languages*, pages 284-285, Boulder, Colorado, September 1996.

Annotation: Describes a system to provide web access to large amounts of heterogeneous information without requiring relocation or restructuring.

[Mishra 1998] Sunanda Mishra. A formal proof of soundness and completeness of the type inference system in Forms/3. Master's thesis, Oregon State University, 1998.

[Mitchell et al. 1995] Richard Mitchell, John Howse, and Ian Maung. As-a: a relationship to support code reuse. *Journal of Object-Oriented Programming*, pages 25-33+, 1995.

Annotation: Distinguishes between "is-a" (classification relationships) and "as-a" (code reuse relationship) and points out the need for programming languages that support this distinction.

[Mitchell 1990] John C. Mitchell. Toward a typed foundation of method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 109-124, San Francisco, California, January 1990.

[Moher 1988] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849-857, June 1988.

Annotation: A pioneering visual debugging and visualization environment for a simplified C-like language. It does not deal with error handling and is oriented toward debugging-in-the-small, but it does provide support for the programmer to observe and control program execution and to interactively create data visualizations that are automatically updated. The programmer need not explicitly program graphics at a low level. Graphics and animations must be in a separate window, separating them from program code.

[Mukherjea and Stasko 1993] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *International Conference on Software Engineering*, pages 456-465, Baltimore, Maryland, May 1993.

Annotation: The Lens system is a prototype software visualization system meant to aid in program tracing, debugging and understanding rather than instruction. Supports rapid development of animations as a problem solving tool for programmers. Geared toward program exploration and high-level debugging.

[Mulet and Cointe 1993] Philippe Mulet and Pierre Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 128-144. First JSSST International Symposium, November 1993.

[Mulholland and Watt 1998] Paul Mulholland and Stuart Watt. Hank: A friendly cognitive modelling language for psychology students. In *Proceedings IEEE Symposium on Visual Languages*, pages 210-216, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Cognitive modeling is the overlap between AI and psychology and involves building computational models of psychological theories in order to learn more about them. Hank is a VPL designed for use by psychology students studying cognitive modeling. The design requirements of Hank included that the language be appropriate for psychology students, suitable for the non-programmer, suitable for working in groups, good at showing the execution path, and usable on paper. The conference presentation of this paper was extremely good and included comments (mostly positive) from students who had used Hank.

[Myers and Vander Zanden 1992] Brad A. Myers and Brad Vander Zanden. Environment for rapidly creating interactive design tools. *The Visual Computer: International Journal of Computer Graphics*, 8(2):94-116, 1992.

Annotation: Overview of different Garnet tools.

[Myers et al. 1990] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71-85, November 1990.

[Myers et al. 1992] Brad A. Myers, Dario A. Giuse, and Brad Vander Zanden. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 182-200, 1992.

[Myers et al. 1997] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alerrency, Andrew Faulring, and others. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347-365, June 1997.

Annotation: Amulet is a user interface development environment for C++. It is Garnet's successor.

[Myers 1991] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *CHI Proceedings: Human Factors in Computing Systems*, pages 243-249, May 1991.

[Najork and Golin 1990] Marc A. Najork and Eric Golin. Enhancing Show-and-Tell with a polymorphic type system and higher order functions. In *Proceedings IEEE Workshop on Visual Languages*, pages 215-220, 1990.

Annotation: ESTL's types provides a visual syntax for Milner's type system.

[Najork 1996] M. Najork. Programming in three dimensions. *Journal of Visual Languages and Computing*, 7(2):219-242, June 1996.

[Ng and Luk 1995] K. W. Ng and C. K. Luk. I^+: A multiparadigm language for object-oriented declarative programming. *Computer Languages*, 21(2):81-100, 1995.

Annotation: Includes object-oriented, logic and functional paradigms in one language. An interesting feature of the language I^+ is its approach to inheritance that allows fine-grained method-by-method inheritance from other objects.

[Nipkow and Prehofer 1993] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *ACM Symposium on Principles of Programming Languages*, pages 409-418, Charleston, South Carolina, January 1993.

Annotation: They claim to have the simplest algorithm published so far that extends ML-style type inference to type classes. They claim soundness and completeness and show the existence of principal types. They allow types to belong to more than one class, where classes are not ordered, and show that subclasses are merely syntactic sugar and can be eliminated. Also includes a brief discussion of ambiguity which affects most type systems with overloading.

[Norman 1989] Don A. Norman. *The Design of Everyday Things*. Doubleday, New York, 1989.

Annotation: Discusses what makes things useful and how everyday items are (or are not) designed for usability.

[North and Koutsofios 1994] Stephen C. North and Eleftherios Koutsofios. Applications of graph visualization. In *Graphics Interface '94*, pages 235-245, May 1994.

Annotation: More information about the Graphviz tools is available from AT&T Research at `http://www.research.att.com/sw/tools/graphviz/`.

[Novak et al. 1992] G. S. Novak, F. N. Hill, M. L. Wan, and B. G. Sayrs. Negotiated interfaces for software reuse. *IEEE Transactions on Software Engineering*, 18(7):646-653, 1992.

Annotation: Subroutine interfaces are rigid, making them difficult to reuse. LINK is a tool that automatically writes an interface conversion program given a subroutine and calling program that both use GLISP types. NI uses menu-based negotiation with user input to generate customized matching procedures from generic ones. Both tools help to reuse components quickly and without requiring a detailed understanding of the data representations of the reused component.

[Nuchprayoon and Korfhage 1997] Assadaporn Nuchprayoon and Robert Korfhage. GUIDO: Visualizing document retrieval. In *Proceedings IEEE Symposium on Visual Languages*, pages 184-188, September 1997.

[Odersky and Wadler 1997] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practise. In *ACM Symposium on Principles of Programming Languages*, January 1997.

Annotation: Pizza, a superset of Java, incorporates three ideas from the academic community: parametric polymorphism, higher-order functions and algebraic data types.

[Olston et al. 1998] Chris Olston, Michael Stonebraker, Alexander Aiken, and Joseph M. Hellerstein. VIQING: Visual Interactive QueryING. In *Proceedings IEEE Symposium on Visual Languages*, pages 162-169, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Visual incremental query on top of visual database display. The described techniques work pretty well for the examples given, but I wonder how generalizable they are.

[Palsberg and Schwartzbach 1991] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 146-161, October 1991.

Annotation: Presents a type inference algorithm for a simplified Smalltalk that includes inheritance, state modification, late binding and polymorphic methods. It guarantees all messages are understood and provides type annotations for instance variables and methods. Inheritance is handled by expanding all classes before doing inference. Conditional type constraints are solved by least fixed-point derivation rather than unification.

[Pandey and Burnett 1993] R. Pandey and M. Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *Proceedings IEEE Symposium on Visual Languages*, pages 344-351, Bergen, Norway, 1993.

Annotation: A study comparing construction of matrix programs in three languages: OSU-APL (APL with English-like syntax), Pascal and Forms/3.

[Penz and Wollinger 1993] Franz Penz and Thomas Wollinger. The ObjectWorld, a classless, object-based, visual programming language. *OOPS Messenger*, 4(1):26-35, January 1993.

Annotation: For concreteness, every object conceptually contains all its data and methods (no inheritance or delegation). Data abstraction is enforced. Polymorphism is achieved through dynamic binding. Reuse is enabled by cloning objects and subparting with automatic message propagation. Once locked, objects can be reused but not modified. For simplicity, there are no types.

[Penz 1991] Franz Penz. Visual programming in the ObjectWorld. *Journal of Visual Languages and Computing*, 2:17-41, 1991.

Annotation: New objects are constructed by direct manipulation of prefabricated objects. A short example of programming in ObjectWorld, a stopwatch, is explained in text and figures.

[Perrone and Repenning 1998] Corrina Perrone and Alexander Repenning. Graphical rewrite rule analogies: Avoiding the inherit or copy & paste reuse dilemma. In *Proceedings IEEE Symposium on Visual Languages*, pages 40-46, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Describes graphical rewrite rule analogies that facilitate reuse in rule-based languages such as Agentsheets. For example, the rule "cars move on roads like trains move on tracks" (expressed visually by the programmer) generates a set of rules for cars and roads that mirrors the rules for trains and tracks. Currently, this is a one-time editing shortcut, although they have plans to explore retaining the connection between the source and copy.

[Peterson and Jones 1993] John Peterson and Mark Jones. Implementing type classes. In *ACM Conference on Programming Language Design and Implementation*, pages 227-236, Albuquerque, New Mexico, June 1993.

Annotation: Good intro material and definitions of terms. They explain type classes with Haskell examples, and most of the paper is devoted to implementing type

classes, although they do briefly discuss extending ML style type inference to support type classes. Also discussed are dictionary conversion and monomorphic restriction.

[Peterson et al. 1997] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, et al. Haskell 1.4: a non-strict, purely functional language. `http://haskell.systemsz.cs.yale.edu/report`, April 1997.

Annotation: An updated version of Hudak et al. "Report on the Programming Language Haskell, A Non-strict, Purely Functional Language," *ACM SIGPLAN Notices*, May 1992.

[Peterson 1994] John Peterson. Dynamic typing in Haskell. Technical Report YALEU/DCS/RR-1022, Yale University, 1994.

[Pfeiffer 1995] Joseph J. Pfeiffer, Jr. Ludwig2: Decoupling program representations from processing models. In *Proceedings IEEE Symposium on Visual Languages*, pages 133-139, Darmstadt, Germany, September 1995.

Annotation: Ludwig2 uses different visual representations in order to enhance readability. Pointers are arrows, mathematical equations are equations.

[Piersol 1986] K. Piersol. Object oriented spreadsheets: The Analytic Spreadsheet Package. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 385-390, 1986.

[Pirolli et al. 1996] Peter Pirolli, James Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the web. In *CHI Proceedings: Human Factors in Computing Systems*, pages 118-125, 1996.

Annotation: Web pages are not highly structured or typed, so they propose some ways to extract structure and type information about a locality of Web pages. The data they extract consists of (1)topology of hyperlinks among pages (2)meta-information such as file size and URL (3)Usage frequency and paths (4)textual similarity. Page types are determined by weighting attributes and ranking the top pages for each category. They suggest using this data for extracting groups of pages suitable for a WebBook [same proceedings].

[Poswig and Morara 1993] Jörg Poswig and Claudio Morara. Incremental type systems and implicit parameter overloading in visual languages. In *Proceedings IEEE Symposium on Visual Languages*, pages 126-133, Bergen, Norway, August 1993.

Annotation: Explains the incremental type inference system of VisaVis. See [Poswig et al. 1994] for more language description.

[Poswig et al. 1992] J. Poswig, K. Teves, G Vrankar, and C. Moraga. VisaVis-contributions to practice and theory of highly interactive visual languages. In *Proceedings IEEE Workshop on Visual Languages*, pages 155-162, Seattle, WA, September 1992.

Annotation: VisaVis allows polymorphic functions, but no user-defined types.

[Poswig et al. 1994] Jörg Poswig, Guido Vrankar, and Claudio Morara. VisaVis: a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, 5(1):83-111, March 1994.

Annotation: VisaVis is a functional visual language supporting higher-order functions. The authors introduce interactive substitution as a means to provide an easy learning metaphor and to prevent syntactical errors. Only briefly mentions type inference.

[Poulin 1993] Jeffrey S. Poulin. Issues in the development and application of reuse metrics in a corporate environment. In *Proceedings International Conference on Software Engineering and Knowledge Engineering*, pages 258-262, San Francisco, California, June 1993.

[Poulin 1995] Jeffrey S. Poulin. Measuring the level of reuse in object-oriented development. In *Workshop on Institutionalizing Software Reuse*, St. Charles, Illinois, August 1995.

[Pree 1995] Wolfgang Pree. Framework development and reuse support. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 12. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

Annotation: Describes visual tools (design book, active cookbook) for reusing frameworks.

[Price and Demurjian Sr. 1997] Margaretha W. Price and Steven A. Demurjian Sr. Analyzing and measuring reusability in object-oriented designs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 22-33, October 1997.

Annotation: Presents a technique for analyzing and measuring the reusability of an OO design. Also gives guidelines for ways to improve a design once analyzed.

[Prieto-Díaz and Freeman 1987] Rubén Prieto-Díaz and Peter Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6-16, January 1987.

Annotation: Suggests an environment that helps locate components and estimates the adaptation and conversion effort necessary for reuse. The user provides a set of functional specifications; 'similar' components are those that match some but not all the requirements. Similar components are ranked according to degree of match and effort to adapt. Rather than an enumerative classification (that divides everything into successively narrower classes, including all compound classes, arranged in a hierarchy), they use faceted classification (from library science) that builds up (synthesizes) compound classes from elemental ones. They describe software functionality with <function, object, medium> and its environment with <system type, functional area, setting> using a controlled vocabulary to avoid duplicate and ambiguous descriptors. Examples are <add, characters, table, relational DB, accounts payable, advertising> and <compress, files, disk, file handler, DB management, catalog sales>. A weighted graph models the conceptual closeness of terms. Three evaluations were performed that seemed to indicate success in retrieval (as compared to a database system); ease of use, accuracy and consistency of classification scheme; and estimation of reuse effort. Open questions mentioned: standardization of classification schedules, validation of reuse effort metrics.

[Prieto-Díaz 1989] Rubén Prieto-Díaz. Classification of reusable modules. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability: Concepts and Models*, volume 1, chapter 4. Addison-Wesley, Reading, Massachusetts, 1989. Based on [Prieto-Díaz and Freeman 1987].

[Prieto-Díaz 1996] R. Prieto-Díaz. Reuse as a new paradigm for software development. In M. Sarshar, editor, *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*. Springer-Verlag, 1996.

Annotation: Reuse in software is not the same as in other areas; a better analogy for software reuse is an assembly line. When software components are harder to create and easier to buy, then the software engineer will generate a good design (not the "best") forcing some compromises on the requirements. We can't change the pay-off of reusing a component, but we can change how easy it is to reuse. What we want, then, are large, easy to reuse components, and higher-level components. Also, effective reuse must be systematic and involve a reuse infrastructure, in other words, there is much more to it than having a repository of reusable components, no matter how easy they are to reuse. Systematic reuse requires an infrastructure, domain analysis, derivation of domain models (for our particular product), process definition and performance measures.

[Rader et al. 1998] Cyndi Rader, Gina Cherry, Cathy Brand, Alexander Repenning, and Clayton Lewis. Designing mixed textual and iconic programming languages for novice users. In *Proceedings IEEE Symposium on Visual Languages*, pages 187-194, Halifax, Nova Scotia, Canada, September 1998.

Annotation: More of a case study about experience using a VPL with children and the resulting improvements made to the language design. They list five design guidelines for program comprehension and composition that they discovered during their experience.

[Raj and Levy 1989] R. K. Raj and H. M. Levy. A compositional model for software reuse. *The Computer Journal*, 32(4), 1989.

Annotation: Emerald supports object encapsulation, subtyping (substitutability), complete separation of type and implementation, locality of definition (no implementation inheritance, so no "hidden parts" of object). The Smalltalk-like browser (Jade) allows textual component location based on categories (components may belong to as many categories as are applicable), synonyms (programmer-provide), role (operation, function, object, type), conformity (which components are substitution compatible), clients (where is this component used), and sub-components.

[Raj et al. 1991] Rajendra K. Raj, Ewan Tempero, and Henry M. Levy. Emerald: A general-purpose programming language. *Software-Practice and Experience*, 21(1):91-118, January 1991.

Annotation: Describes Emerald, comparing it to other programming languages. Emerald embraces the concept of self-sufficient objects.

[Raj 1991] Rajendra K. Raj. Composition and reuse in object-oriented languages. Technical Report 91-03-06, University of Washington, March 1991. Reformatted version of author's Ph.D. dissertation.

Annotation: Discusses metrics for measuring reuse in object-oriented languages, the difference between conformance (shared object interfaces) and inheritance (shared object implementations), and reuse via composition using the Jade model. "For reuse to be effective, the Jade language constructs must be complemented by programming environment support." (Sec 4.4).

[Redmiles 1993] David F. Redmiles. Reducing the variability of programmers' performance through explained examples. In *CHI Proceedings: Human Factors in Computing Systems*, pages 67-73, Amsterdam, The Netherlands, April 1993.

Annotation: An empirical study demonstrating that explained examples are better than traditional on-line help facilities. Subjects using the EXPLAINER used less trial-and-error and performed the task more directly.

[Repenning and Ambach 1996] Alexander Repenning and James Ambach. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *Proceedings IEEE Symposium on Visual Languages*, pages 102-109, Boulder, Colorado, September 1996.

Annotation: Programmed agents can be placed on a web-based repository and dragged onto other worksheets.

[Repenning and Ioannidou 1997] Alexander Repenning and Andri Ioannidou. Behavior processors: Layers between end-users and Java virtual machines. In *Proceedings IEEE Symposium on Visual Languages*, pages 402-409, September 1997.

Annotation: Introduces the analogy of Java as a "Postscript" for programs. Ristretto is a tool for turning Agentsheet programs into Java applets.

[Risley and Smedley 1998] Christopher C. Risley and Trevor J. Smedley. Visualization of compile time errors in a Java compatible visual language. In *Proceedings IEEE Symposium on Visual Languages*, pages 22-29, Halifax, Nova Scotia, Canada, September 1998.

Annotation: Using a Prograph-like syntax for Java semantics, compile time errors are made visible at edit time. "Used before set" errors are highlighted (and propagated), type errors cause dataflow lines to appear dotted, exceptions are handled with little icons and highlighting.

[Robbins et al. 1996] Jason E. Robbins, David J. Morley, David F. Redmiles, Vadim Filatov, and Dima Kononov. Visual language features supporting human-human and human-computer communication. In *Proceedings IEEE Symposium on Visual Languages*, pages 247-254, Boulder, Colorado, September 1996.

Annotation: Describes a diagram-based visual object-oriented language for designing and programming visual simulations of factories. Interesting aspects include multiple, overlapping views (rather than hierarchical views) and customizable presentation graphics. They use the metaphor of machines with push-buttons for both direct manipulation and programming.

[Rosson and Carroll 1996] Mary Beth Rosson and John M. Carroll. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3):219-253, September 1996.

Annotation: The authors observed heavy use of previously-existing usage contexts by Smalltalk programmers.

[Rüger et al. 1996] Michael Rüger, Bernhard Preim, and Alf Ritter. Zoom navigation: Exploring large information and application spaces. In *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, pages 40-48, Gubbio, Italy, May 1996.

Annotation: Adds aspect-of-interest (AOI) function to the degree-of-interest (DOI) usually associated with fish-eye views.

[Schiffer and Fröhlich 1995] Stefan Schiffer and Joachim Hans Fröhlich. Visual programming and software engineering with vista. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 10. Prentice-Hall/Manning Publications Co., Greenwich, CT, 1995.

Annotation: Vista is a visual object-oriented language that strives to achieve such software engineering principles as weak coupling between building blocks, procedural and data abstraction, safety through static type checking, and component reuse (and direct integration of documentation). Although class-based, the programmer first constructs a concrete prototypical object from which to generate the class definition. One of the consequences of its integration with Smalltalk is that some information hiding enforced in Vista can be circumvented in low-level Smalltalk code. The type system uses available information to detect and prevent obvious type incompatibilities, but does not guarantee the absence of run-time type errors.

[Schmidt and Omohundro 1993] Heinz W. Schmidt and Stephen M. Omohundro. CLOS, Eiffel, and Sather: A comparison. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 181-213. MIT Press Cambridge, Massachusetts, London, England, 1993.

Annotation: Discusses the approaches taken by each of the three languages with respect to many object-oriented (and more general) language issues such as multiple inheritance, the relationship between classes and types, garbage collection and language environment. (Also available as International Computer Science Institute TR-91-047).

[Schmidt 1995] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65-74, October 1995.

Annotation: Describes an example design pattern, the Reactor, and discusses lessons learned from using design patterns to develop OO communication frameworks for several different projects. These lessons include: patterns enabled reuse of high-level componentry—the software architecture, pattern descriptions should contain concrete examples for better understandability, patterns improve communication by providing a shared vocabulary and high-level concepts about essential properties of the software, pattern names should be chosen carefully and used consistently (more verbose aliases can help), patterns are no substitute for design and implementation, patterns are validated by experience rather than testing, pattern descriptions contain explicit documentation of tradeoffs and alternatives and where the pattern does and does not apply, patterns facilitate new developer training (project understanding) by giving a big-picture view, managing expectations is crucial to using patterns effectively.

[Schmidt 1996] David A. Schmidt. Programming language semantics. In Allen Tucker, editor, *CRC Handbook of Computer Science*. CRC Press, 1996. Summary of ACM Computing Surveys article, 28(1), 1996.

Annotation: A survey of programming language semantics.

[Schwartzbach 1997] Michael I. Schwartzbach. Object-oriented type systems: Principles and applications. Lecture notes available at `http://www.daimi.aau.dk/~mis/ootspa.ps`, May 1997.

[Sebesta 1996] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Menlo Park, CA, third edition, 1996.

[Seidewitz 1996] Ed Seidewitz. Controlling inheritance. *Journal of Object Oriented Programming*, 8(8):36-42, January 1996.

Annotation: Advocates the use of abstract classes for interface inheritance combined with mixin classes for implementation inheritance. Addresses the problems of classification tied to implementation, rigid hierarchy, and distribution of object definitions (yo-yo problem).

[Selby 1989] Richard W. Selby. Quantitative studies of software reuse. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability: Applications and Experience*, volume 2. Addison-Wesley, Reading, Massachusetts, 1989.

[Selfridge and Srivastava 1996] Peter Selfridge and Divesh Srivastava. A visual language for interactive data exploration and analysis. In *Proceedings IEEE Symposium on Visual Languages*, pages 84-85, Boulder, Colorado, September 1996.

Annotation: The IDEA language can be viewed as a dataflow diagram or a history of actions. A diagram can be saved for later reuse.

[Sengupta et al. 1994] Samudra Sengupta, Takayuki Dam Kimura, and Ajay Apte. An artist's studio: A metaphor for modularity and abstraction in a graphical diagramming environment. In *Proceedings IEEE Symposium on Visual Languages*, pages 128-136, St. Louis, Missouri, October 1994.

Annotation: Supports sharing of objects in a graphical editor meant to be parsed as a visual language. Suitable for any node-and-wire VPL that consists of a graphical diagramming environment and an interpreter.

[Shizuki et al. 1998] Buntarou Shizuki, Masashi Toyoda, and Esuya Shibayama. Visual patterns + multi-focus fisheye view: An automatic scalable visualization technique of data-flow visual program execution. In *Proceedings IEEE Symposium on Visual Languages*, pages 270-277, Halifax, Nova Scotia, Canada, September 1998.

Annotation: A technique for automatic animations of dataflow VPL execution. Includes a framework for browsing different animations that highlight different aspects of the program. They use both fisheye techniques and semantic zooming. Based on the visual design patterns from the authors' VL'97 paper.

[Shneiderman 1996] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings IEEE Symposium on Visual Languages*, pages 336-343, Boulder, Colorado, September 1996.

Annotation: The author proposes a taxonomy of information visualizations of large amounts of data. The designer's goal should be comprehension, control and responsibility, avoiding confusion, frustration and remorse. Seven data types are listed: one-dimensional, two-dimensional, three-dimensional, temporal, multi-dimensional, trees, and networks. The user should be able to perform all seven tasks in a continuous, visual manner: overview, zoom, filter, details-on-demand, relate, history, and extract.

[Sitaraman 1996] Murali Sitaraman, editor. *Proceedings of the IEEE Conference on Software Reuse*, Orlando, FL, April 1996.

[Slonneger and Kurtz 1995] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: a laboratory based approach.* Addison-Wesley Publishing Company, Inc., 1995.

[Smedley et al. 1996] T. Smedley, P. Cox, and S. Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Advanced Visual Interfaces*, pages 148-155, May 1996.

[Smith et al. 1994] Randall B. Smith, Mark Lentczner, Walter R. Smith, Antero Taivalsaari, and David Ungar. Prototype-based languages: Object lessons from class-free programming. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Panel summary appeared in the proceedings addendum, 1994.

Annotation: Panelists talked about the good and bad aspects of various prototype-based languages. One theme that seemed to emerge was the problem of operating on groups of objects or a notion of "collectionness" for prototypes.

[Smith 1995] Walter Smith. Using a prototype-based language for user interface: The Newton project's experience. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 61-72, Austin, Texas, October 1995. Published as SIGPLAN Notices 30(10).

Annotation: Describes NewtonScript, a prototype-based object oriented language designed for user interface construction. Objects can inherit from both a prototype and a parent (container inheritance). The result is a very flexible programming language; they indicate that they would consider enforcing more program structure through either the language or environment. They end with a plea for more research on prototype-based languages and programming environments.

[Sondergaard and Sestoft 1990] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505-517, 1990.

Annotation: Definitions of referential transparency (preserving substitutivity of identity) and closely related concepts.

[Spenke et al. 1996] Michael Spenke, Christian Beilken, and Thomas Berlage. FOCUS: The interactive table for product comparison and selection. In *ACM Symposium on User Interface Software and Technology*, pages 41-50, Seattle, Washington, November 1996.

Annotation: need to do.

[Stasko and Muthukumarasamy 1996] John Stasko and Jeyakumar Muthukumarasamy. Visualizing program executions on large data sets. In *Proceedings IEEE Symposium on Visual Languages*, pages 166-173, Boulder, Colorado, September 1996.

Annotation: They describe techniques for visualizing programs working on large data sets where the data outnumbers the display pixels. Graphical objects in the display can represent individual values or cumulative or clustered data. Semantic zooming is defined as including an overview (usually abstracted) of all data in one window without scrolling. Users interactively zoom in on a portion of the program data by selecting a representative graphical object. At the most detailed view, the visualization uses recognized algorithm animation or program visualization presentation styles.

[Stasko et al. 1993] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *CHI Proceedings: Human Factors in Computing Systems*, pages 61-66, Amsterdam, The Netherlands, April 1993.

Annotation: Found no significant benefit from the sort animations used for instruction. Their conjectures: animations must be motivated by comprehensive motivational instruction, must be geared to specific instructional goals, and should include rewind-replay capabilities.

[Stasko 1990] John T. Stasko. Simplifying algorithm animation with TANGO. In *Proceedings IEEE Workshop on Visual Languages*, pages 1-6, Skokie, Illinois, 1990.

[Stein et al. 1989] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 3, pages 31-48. ACM Press, Addison-Wesley, 1989. Also in SIGPLAN Notices 23(5).

Annotation: OO is not a dichotomy of class/inheritance languages and prototype/delegation languages. Rather OOLs are characterized by variations on two fundamental mechanisms, templates (for creating new objects like old ones) and empathy (allowing an object to act as if it were another, thus sharing state and behavior). The variations fall into three independent dimensions: is the sharing defined statically or dynamically, implicitly or explicitly, and per object or per group? "The true value of object-oriented techniques as opposed to conventional programming techniques is not that they can do things the conventional techniques can't, but that they can often extend behavior by adding new code in cases where conventional techniques would require editing existing code." (p. 44).

[Stroustrup 1992] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1992.

[Stroustrup 1996] B. Stroustrup. Keynote address: Language-technical aspects of reuse. In *International Conference on Software Reuse*, pages 11-19, Orlando, Florida, April 1996.

Annotation: A PL can simplify the expression of clean design and provide facilities for tailoring components (class hierarchy, type parameterization).

[Szypersky and Omohundro 1993] Clemens Szypersky and Stephen Omohundro. Engineering a programming language: The type and class system of Sather. Technical Report TR-93-064, International Computer Science Institute, `http://www.icsi.berkley.edu/`, November 1993.

[Taivalsaari 1993] Antero Taivalsaari. *A critical view of inheritance and reusability in object-oriented programming*. Ph.D. thesis, University of Jyväskylä, 1993.

Annotation: Detailed discussions of inheritance and reuse and introduction to Kevo. Kevo is a prototype-based language emphasizing concreteness and self-sufficiency of objects. Objects are created by cloning an existing object. Objects are true individuals, and can be modified on an individual basis. Groupwise modifications are made possible by *clone families* , which are automatically managed by the system. A small syntactic reminder (a "*") indicates whether an operation should apply to the entire family rather than the individual object. Thus Kevo does not

require a designated parent prototype for a collection of objects. Similar objects that do not belong to the same clone family still must be modified separately, however, and it is not clear that the clone families automatically inferred by the system are equivalent to the families the programmer has in mind. Kevo approximates multiple inheritance and fine-grained inheritance via a cut/copy/paste metaphor, but it is not truly inheritance since changes to the original code do not propagate, but must be recopied by the programmer. To facilitate program structuring, *facets* , partial objects used as building blocks for other objects (similar to mixins or abstract classes) are an optional programming technique.

[Tanimoto 1990] S. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 2(2):127-139, June 1990.

Annotation: Includes definition of liveness and a scale from 1 to 4. A language at level 1 liveness provides no semantic feedback without compilation. At level 2 the user can obtain semantic feedback on request (as in 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 other events such as system clock ticks.

[Tracz 1987] Will Tracz. Reusability comes of age. *IEEE Software*, pages 6-8, July 1987.

Annotation: Reusing software is like buying a used car—an extended analogy to explain why some are leery of used software. Successful used-software needs quality parts, standard interfaces, documentation, choice of options.

[Trazc 1995] Will Trazc. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.

[Ungar and Smith 1987] David Ungar and Randall Smith. Self: the power of simplicity. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227-241, October 1987.

[Ungar et al. 1991a] David Ungar, Craig Chamber, Bay-Wei Chang, and Urs Hölze. Organizing programs without classes. *Journal of Lisp and Symbolic Computation*, 4(3), June 1991.

Annotation: Compares class-based organization to Self's prototype organization. Organization can be achieved by splitting the implementation of an object into two separate objects; in Self, a traits object holds shared behavior and data.

[Ungar et al. 1991b] David Ungar, Craig Chamber, Bay-Wei Chang, and Urs Hölze. Parents are shared parts of objects: Inheritance and encapsulation in S*elf. Journal of Lisp and Symbolic Computation*, 4(3), 1991.

Annotation: Describes inheritance, especially multiple inheritance, of Self and the mechanisms they use for resolving ambiguities. Encapsulation and the meaning of public, private and unspecified slots are also discussed.

[Ungar 1995] David Ungar. Annotating objects for transport to other worlds. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 73-87, Austin, Texas, October 1995. Also in SIGPLAN Notices 30(10).

Annotation: The authors claim that five missing pieces of information are needed for saving directly-constructed programs: which module to use, which value (actual or initial) to save, whether to create a new object in a new world or refer to an existing one, whether an object is immutable with respect to transportation, and whether the save should be low-level or an abstract type-specific expression.

[Vaishnavi and Bandi 1996] Vikay K. Vaishnavi and Rajendra K. Bandi. Measuring reuse. *Object Magazine*, 6(2), April 1996.

Annotation: Extends the Goal-Question-Metric (GQM) model to the Framework Assisted GQM Model that helps establish perspective when measuring reuse.

[van Zee et al. 1996] Pieter van Zee, Margaret Burnett, and Maureen Chesire. Retire Superman: Handling exceptions seamlessly in a declarative visual programming language. In *Proceedings IEEE Symposium on Visual Languages*, pages 222-230, Boulder, Colorado, September 1996.

Annotation: "Bad clock" and factorial examples of exception handling in Forms/3. All accomplished without added language constructs.

[Vion-Dury and Pacull 1997] J.-Y. Vion-Dury and F. Pacull. A structured interactive workspace for a visual configuration language. In *Proceedings IEEE Symposium on Visual Languages*, pages 130-137, Capri, Italy, September 1997.

Annotation: Type inference applied to graphical types and used for rendering purposes. Composite glyphs infer attributes from their subcomponents.

[Walpole and Burnett 1997] Rebecca A. Walpole and Margaret M. Burnett. Supporting reuse of evolving visual code. In *IEEE Symposium on Visual Languages*, pages 68-75, September 1997.

Annotation: Forms/3 is used to prototype support for code reuse in VPLs with informal repositories (ones without an owning institution enforcing standards or producer packaging of code), such as one might find on the Web. Identifies ways VPL features can be leveraged to provide answers to common reuse questions. (What's evolving is the repository, we don't handle versioning, although that would be a nice addition.).

[Walpole and Cook 1996] Rebecca A. Walpole and Curtis Cook. Software reuse primer. Technical Report 96-60-16, Oregon State University, June 1996.

[Wand and O'Keefe 1991] Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In J. L. Lassez and G. D. Plotkin, editors, *Computational Logic: in honor of J. Alan Robinson*, pages 479-486. MIT Press, 1991.

[Wand 1986] Mitchell Wand. Finding the source of type errors. In *ACM Symposium on Principles of Programming Languages*, pages 38-43, St. Petersburg Beach, Florida, January 1986.

Annotation: Presents a modified Boyer-Moore unification algorithm to keep track of reasons for type inferences made. Type errors reported to the compiler include these reasons, one of which should be the actual source of the error.

[Wand 1987] Mitchell Wand. Complete type inference for simple objects. In *Proceedings IEEE Symposium on Logic in Computer Science*, pages 37-44, 1987. Corrigendum, *Proceedings 3rd IEEE Symposium on Logic in Computer Science*, page 132, 1988.

Annotation: Simple objects are modeled with records. The unification algorithm in this paper was later found to be incorrect. See [Wand 1994] for his later work.

[Wand 1994] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97-120. MIT Press, 1994. Originally appeared as Technical Report NU-CCS-89-2, Northeastern University College of Computer Science, February 1989.

Annotation: Type inference for objects with protected instance variables, public methods, single inheritance and first-class classes. Accomplished by extending polymorphic record typing (Rèmy) to allow infinite label sets and modeling objects in this new language. Appendix includes ML code for the unification algorithm. Results in a correct program or a "Unify Failed" message.

[Wand 1993] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1-15, 1993.

Annotation: Introduces a lambda calculus with records, including record concatenation operator. An example of OO programming including hidden instance variables and multiple inheritance may be coded in this calculus.

[Wang and Ambler 1994] Guijun Wang and Allen Ambler. Applicability checking in visual programming languages. In *Proceedings IEEE Symposium on Visual Languages*, pages 31-38, St. Louis, Missouri, October 1994.

Annotation: Automatic detection of boundary problems and assistance in resolving the formulas at the boundaries of matrices.

[Wang and Ambler 1995] Guijun Wang and Allen Ambler. Invocation polymorphism. In *Proceedings IEEE Symposium on Visual Languages*, pages 83-90, Darmstadt, Germany, September 1995.

Annotation: The authors define invocation polymorphism as an abstraction not on the definition of a function, but on its invocation (one programmed invocation may result in several invocations, for example). The system automatically analyzes structures of arguments to determine how a function should be invoked.

[Wang and Ambler 1996] Guijun Wang and Allen Ambler. Solving display-based problems. In *Proceedings IEEE Symposium on Visual Languages*, pages 122-129, Boulder, Colorado, September 1996.

Annotation: A follow-up to David Hendry's VL'95 paper [Hendry 1995].

[Wegner 1987] Peter Wegner. Dimensions of object-based language design. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 168-182, Orlando, Florida, 1987. Also in SIGPLAN Notices 22(12).

Annotation: Includes definitions of object-based and object-oriented languages. Emphasizes class-based mechanisms.

[Weiser 1984] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.

Annotation: Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow.

[Whitley 1997] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8:109-142, 1997.

Annotation: Excellent summaries of empirical studies involving visual languages and visual representations. Includes evidence both for and against the usefulness of VPLs.

[Wilcox et al. 1997] E. Wilcox, J. Atwood, M. Burnett, J. Cadiz, and C. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *CHI Proceedings: Human Factors in Computing Systems*, pages 258-265, Atlanta, GA, March 1997.

Annotation: Continuous visual feedback did not help with debugging in general, but did aid accuracy for some situations. Significant differences were also noted with respect to behavior. Results also seemed to vary among three factors: type of user, type of bug, and type of problem.

[Wilde and Lewis 1990] N. Wilde and C. Lewis. Spreadsheet-based interactive graphics: From prototype to tool. In *CHI Proceedings: Human Factors in Computing Systems*, pages 153-159, April 1990.

[Wilde et al. 1993] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, pages 75-80, January 1993.

Annotation: Some aspects of good object-oriented design can make maintenance for OO programs difficult. Inheritance can obscure calling and dataflow relationships, making these dependencies harder to find and analyze. Program functions may not be in one place, but rather distributed among groups of cooperating objects.

[Wirth 1988] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204-214, April 1988.

Annotation: Data types are treated as extensible records. This allows reuse of types defined in other modules without accessing the source code of that module. Extended types can be used in place of the original type (substitutability, polymorphism). The mechanism is purely composition; there is no overriding, selective inclusion, etc.

[Wittenburg and Sigman 1997] Kent Wittenburg and Eric Sigman. Visual focusing and transition techniques in a Treeviewer for web information access. In *Proceedings IEEE Symposium on Visual Languages*, pages 20-27, 1997.

Annotation: A browser for web statistics. Includes context vs. detail mechanisms (multifocus fisheye), direct manipulation with aggregate capabilities for collapsing/expanding search results, visual feedback regarding relevance, multiple foci and simple layouts for fast performance (table of contents format). A 4-step animation mechanism for view transitions is discussed in the context of visual cohesion.

[Woodfield et al. 1987] Scott N. Woodfield, David W. Embley, and Del T. Scott. Can programmers reuse software. *IEEE Software*, pages 52-59, July 1987.

Annotation: According to their study, people untrained in reuse performed poorly when asked to evaluate the reusability of data structure components.

[Yang and Burnett 1994] S. Yang and M. M. Burnett. From concrete forms to generalized abstractions through perspective-oriented analysis of logical

relationships. In *Proceedings IEEE Symposium on Visual Languages*, pages 6-14, St. Louis, Missouri, October 1994.

Annotation: The fibonacci example illustrates how formulas are generalized from the concrete examples in Forms/3.

[Yang et al. 1996] Sherry Yang, Elyon DeKoven, and Moshé Zloof. Design benchmarks for VPL static representations. In *Proceedings IEEE Symposium on Visual Languages*, pages 263-264, Boulder, Colorado, September 1996.

Annotation: Short version of [Yang et al. 1997].

[Yang et al. 1997] Sherry Yang, Margaret M. Burnett, Elyon DeKoven, and Moshé Zloof. Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing*, 1997.

Annotation: Identifies 3 categories for the design benchmarks: understandability, scalability and audience (or domain) specific. The benchmarks are designed to help VPL designers discover problems with their designs and compare alternate designs.

[Yang 1996] Sherry Yang. *Generalizing Abstraction in Form-Based Languages: From Direct Manipulation to Static Representation*. Ph.D. thesis, Oregon State University, November 1996.

Annotation: Includes generalization and static representation (design benchmarks).

APPENDIX

Appendix

We used the Representation Design Benchmarks [Yang et al. 1997] to evaluate and improve our design of the similarity inheritance representation introduced in Chapter 4. The design benchmarks record how designed representation relates to various findings from cognitive psychology about information that helps programmers with their programming [Green and Petre 1996]. The benchmarks allow a designer to measure a static representation in nine categories, and to decide based upon these measurements whether changes in the design are needed. To do this, the designer evaluates a draft of the design, computing measurements in each category and subjectively evaluating the measurements to identify possible areas for improvement. Although originally developed for use with an entire language design, we were able to use the benchmarks on the subset of the language design affected by similarity inheritance. As a result of this process, we were able to identify weaknesses in and develop improvements for early versions of our design. Here we describe the evaluation of the final design, using the benchmark notation and terms defined in [Yang et al. 1997].

Visibility of Dependencies

The first visibility of dependencies benchmark, D1, is the ratio of program dependencies that are explicitly visible to the programmer. Green and Petre noted hidden dependencies as a severe source of difficulty in understanding programs. The dependencies of interest in the similarity inheritance model are those created by the $\rightarrow$ relationship. The dependencies can be at either the form or cell level. For each, the programmer may be interested in both "what affects this?" and "what does this affect?" for a total of four kinds of dependencies. Because of the importance of explicit

representation to the similarity inheritance model, it was important that all four kinds of dependencies be explicitly represented so that D1=4/4=1.

The two "what affects this" questions are handled by legends at both the form and cell level (refer back to Figure 4.5). The legends statically show the name of the directly affecting form or cell and (if different) the name of the indirectly affecting form or cell which is the original. The programmer has dynamic access to the list of intermediately affecting forms or cells (if any) and can directly access the forms by clicking on their names, but these interaction details do not affect the navigable static representation to which the benchmarks apply.

The two "what does this affect" questions are answered by optional arrows pointing from a cell to all its copies (refer to Figure 4.5) and by optional arrows in the repository view pointing from a form to all its copies (refer to Figure 4.6). The design also includes dotted arrows in the repository view that point from a form to all forms that have copied at least one cell from it.

The worst case number of steps required to navigate to the display of all dependency information, D2, is made up of four parts:

- cost to show form legends: none, they are always visible
- cost to show cell legends: n/2 where n is the total number of cells on the forms of interest. All formulas can be toggled on in one step, but if the programmer wants to see exactly half of them, the other half must be dismissed individually.
- cost to show copy dependency arrows among cells: n/2 where n is the total number of cells on the forms of interest. The design allows all arrows to be toggled on in one step, but if the programmer wants to see exactly half of them, the other half must be dismissed individually.
- cost to show repository arrows: According to the design, each kind of arrow is either on or off in the repository window, so the worst case number of steps is three, one step to open the repository window plus one step to turn each of the similarity arrows on.

Thus D2=n/2+n/2+3. In practice, we would expect the programmer to be interested in only one chain of dependency over the whole program or all the

dependencies in a small part of the program, so we expect the average cost to be much lower than the worst case.

Visibility of Program Structure

The PS1 benchmark is the answer to the question, "Does the representation explicitly show how the parts of the program logically fit together?" For our subset of the language design, the structure in question is the set of overall inheritance relationships among objects, a higher-level view than for individual cells or forms. The repository overview design contains a view of these relationships, enabling the programmer to see chains of dependency at the form level. This copy dependency view is comparable to a class browser for a class-based language. PS2 is the worst case number of steps required to navigate to the display of the structure. The number of steps required to show the repository view and the two kinds of copy dependency arrows is three (one to show the repository window and one each to turn on the arrows). PS1=yes and PS2=3.

Visibility of Program Logic

Benchmark L1 is whether the representation explicitly shows how an element is computed. Since the design makes an inherited formula visible in every place where it is inherited, the logic that computes a cell's value is always explicitly represented as its own formula. This avoids the "yo-yo problem" in class-based languages where to see the program logic for a subclass, the programmer may need to visit several classes up and down the class hierarchy. The yo-yo problem is also exhibited in most prototype-based languages, where instead of class definitions, the programmer must examine multiple levels of parent objects to view inherited code. Thus L1=yes. The navigation cost benchmark L2=n/2, where n is the number of cells on the forms of interest. This is the same measure as for the design of Forms/3 before adding similarity inheritance.

The third measure for program logic, L3, counts the number of misrepresentations of generality. Generality enters the design only in the representation of a polymorphic reference. If the representation of the formula for the cell Mirror:newPoint in Figure 5.3, for example, were "248-Point:movedPoint" for the copy of form Mirror as well as the original, that would be a misrepresentation of generality because the copy actually references a different form. Instead, the design calls for the generalized formula representation "VADT(oldPoint):like movedPoint." Since there are no misrepresentations of generality in the representation of polymorphic references, L3=0.

Display of Results with Program Logic

Benchmark R1 measures whether results can be displayed statically with program source code. In our design the results of cell formulas, whether inherited or not, are visible via liveness and sample values. Since the program logic (formulas) can be displayed at the same time, R1=yes. Because the results are always visible, the number of steps to display them, R2, is zero. We consider this aspect to be a valuable source of feedback for the programmer since the results of inherited code and its impact in the inheriting context are immediately visible.

Secondary Notation

The addition of similarity inheritance did not remove any of the secondary notation (documentation) available in the language nor add any new categories of secondary notation. However, it is interesting to note that the design does add non-semantic notational devices, in the form of arrows and legends, that are automatic rather than provided by the programmer.

Abstraction Gradient

Before the addition of similarity inheritance, the proportion of details that can be abstracted away (out of four: data details, operation details, other fine-grained details, and controls), which is AG1, was 4/4. The abstraction gradient benchmark is affected by the addition of similarity inheritance in two ways. First, the repository's query interface adds an additional set of controls. These controls can be collapsed into icon form, so the abstraction gradient for control is not adversely affected. Second, the repository view provides an additional mechanism for abstraction of form data. Thus for Forms/3 language design overall, the benchmark AG1 remains 4/4=1 and the navigation benchmark AG2 is one step each for both abstractions provided by the similarity inheritance representation.

Accessibility of Related Information

The RI1 benchmark is whether it is possible to display all related information side by side. Any two forms can be placed side by side, any cells can be moved around on the form so that they appear side by side (or to the edge of a form to appear side by side with a cell on a different form), the repository window can be placed side by side with any form, and nodes within the repository graph can be dragged to any position. Thus RI1=yes since it is possible to display all related similarity information side by side. The number of steps to navigate to related similarity information, RI2, depends on which information is being accessed. Through the form and cell legends, the programmer can access an affecting form or cell in two steps. Cell copies can be found by navigating the copy dependency arrows which require one step to turn on for each cell examined; to navigate to all direct and indirect copies of a cell, the cost is c steps where c is the number of directly and indirectly copied cells. Form copies can be displayed in one step from the repository window.

Screen Real Estate

Whenever features are added to a VPL's representation, the limited size of the screen must be taken into account[1]. Our representation design added several real-estate consuming devices: the repository window, form legends, cell legends, and copy dependency arrows. Since the actual numbers of the benchmarks SRE1, the number of program elements that can be displayed on the physical screen, and SRE2, the number of non-semantic intersections present when obtaining the SRE1 score, are not very meaningful without some context, we will examine how the addition of each of these devices affects the previous use of screen real estate in Forms/3.

The repository adds one or two extra (optional and iconifiable) windows to the programmer's workspace. The default view displays can easily display 30 forms at a time (the actual number displayed depends on the edges between forms; the current default layout average is closer to 16 forms). The repository's navigational devices allow a high degree of programmer control over the trade-off between screen space used and time to locate the needed information. The entire graph can be reduced or increased in size, a birdseye overview can be viewed alongside the normal view and used as a navigation tool, and queries can be used to display just a subgraph of the entire repository. There are very few non-semantic intersections when just the dataflow arrows are displayed. Adding similarity arrows to the design increases the likelihood of arrows crossing, which is why the design includes the ability to toggle on or off each kind of arrow.

The other three representation devices, form legends, cell legends and copy dependency arrows, all take up space within the programmer's code. The display of form legends takes up at most one extra line on the form, no matter how long the

[1]The screen real estate problem is also discussed as a part of the larger VPL problem of scaling up to large, complex programs [Burnett et al. 1995a].

inheritance chain becomes, because intermediate forms are normally elided. Likewise, cell legends take at most one line per formula, which slightly reduces the number of cells that are displayable without non-semantic intersections. Copy dependency arrows take up no extra space on the display, but they may introduce non-semantic intersections as the arrows cross other arrows and cells. The number of intersections varies with the actual dependencies in the program, but increases with the number of cells that have their arrows turned on.

Table A illustrates the trade-offs between amount of information represented and real-estate space used; more features present results in fewer program elements on the screen (when non-semantic intersections are minimized).

Design Options	SRE1 (units = cells)	SRE2 (units = intersections)
Base: Design 1, all formulas showing	36	0
Design 1 + dataflow arrows (if request is for a small number of cells)	36 (no change)	a ($a \geq 0$)
Design 1 + dataflow arrows (if request is for all cells)	36 (no change)	b ($b \geq a \geq 0$)
Design 1 + program structure view	29 (approximately 20% fewer)	c ($b \geq c \geq 0$) These arrows are a more coarse-grained view than the dataflow arrows in the previous row.
Design + legends	18 if each cell has one legend displayed (approximately one fewer per legend)	0
Design 1 + cell icons in formulas	29 (approximately 20% fewer)	0
Design 2 (all of above features)	22 (approximately 40% fewer)	b
Design 1 + form legends	35 (approximately 3% fewer)	0
Design 1 + cell similarity legends	30 (approximately 17% fewer)	0
Design 1 + copy dependency arrows	36 (no change)	d ($d \geq 0$)
Design 1 + repository	25 (approximately 30% fewer)	e ($e \geq 0$)
All similarity inheritance features	22 (approximately 40% fewer)	d + e

Table A Screen real estate benchmarks (expanded from [Yang et al. 1997]). Trade-offs between added features and the real-estate space cost become apparent in this comparison. Yang's Design 2 contains all the features described in [Yang et al. 1997]. Design 1 is the equivalent of turning all the features off. The variables a, b, c, d and e represent numbers of line crossings, and their values vary with the actual dependencies in each program. In each case, the screen layout that minimized SRE2 was chosen. The last five rows expand the comparison to include representations added for similarity inheritance.