Journal of Visual Languages and Computing (2000) **11**, 191–235
doi:10.1006/jvlc.1999.0153, available online at http://www.idealibrary.com on IDEAL®

Static Type Inference for a First-Order Declarative Visual Programming Language with Inheritance

REBECCA WALPOLE DJANG, MARGARET M. BURNETT AND ROGER D. CHEN

Department of Computer Science, Oregon State University, Corvallis, OR 97331, U.S.A.
{djang,burnett,rchen}@cs.orst.edu

Received 18 August 1998; revised 7 June 1999; accepted 5 November 1999

The early detection of type errors is a well-known benefit of static typing, but until recent years, this benefit usually has come at the cost of requiring the programmer to explicitly declare the type of every object in a program. Since many visual programming languages (VPLs), especially those VPLs intended for end users, are designed to eliminate such programming mechanisms, most VPLs have been implemented with dynamic typing, thereby sacrificing early type error feedback and other benefits of static typing. One potential solution for this dilemma is static type inference, but unfortunately, the types inferred under previous approaches have been notoriously difficult to understand, even for professional programmers. Compounding this problem is the fact that when support for inheritance is added to such type inference systems, explicit type declarations have re-emerged.

In this paper, we present a model of types that supports static type inference for a declarative VPL that includes inheritance. Our model addresses the problems presented in the previous paragraph. We present the formal model of our type system, and show that the model is not only sound with respect to type safety, but that it also has sufficient power to support traditional and non-traditional forms of inheritance, and further that it requires the user to understand only a small vocabulary of types, a feature important in addressing the understandability problem in end-user VPLs.
© 2000 Academic Press

1. Introduction

EVEN THOUGH MANY OF TODAY'S VISUAL PROGRAMMING LANGUAGES (VPLs) are designed for programmers, an ever-increasing number is aimed at end users. This paper describes a new model of types, intended to support static type inference in first-order declarative VPLs—declarative VPLs with no higher-order functions, such as rule-based, constraint-based, dataflow, logic, and spreadsheet-based VPLs—aimed at either one of these audiences. We focus on first-order declarative VPLs for two reasons: (1) in textual languages, static type inference has been extensively studied in the realm of declarative languages, a topic that is discussed further in Section 2; and (2) in VPL research, first-order declarative paradigms have been widely used.

Most previous work into static types for VPLs has focused on declarative VPLs, but has not included support for inheritance. This lack is a serious limitation on the ability to employ static types in modern declarative VPLs, many of which are starting to incorporate forms of inheritance. In the context of our own research, the need for a model of types supporting inheritance became paramount in our recent work to develop similarity inheritance [1]. Similarity inheritance is an approach to inheritance that eliminates the concept of subclasses. Without subclasses, 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. Similarity inheritance is a fine-grained approach to inheritance without subclasses devised both for VPLs aimed at programmers and for those aimed at end users. However, because the granularity of similarity inheritance is finer than previous approaches to inheritance, models of types that have been developed for (textual or visual) languages with inheritance do not support it.

In this paper, we present a new model of types to support static type inference in a first-order declarative VPL that includes inheritance. Because one of our primary motivations was the ability to support not only traditional forms of inheritance, but also similarity inheritance, one property required of the model of types is:

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

Another problem has been that previous models of types supporting static type inference have not been well suited to VPLs aimed at end users. Yet, a type inference approach that was 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. One of these benefits is particularly important to VPLs that feature immediate visual feedback: a static approach potentially allows a system to immediately report a type error as soon as the offending expression is entered. Thus, additional goals of this research have been to devise an approach to type inference—suited to VPLs with inheritance, including those for end users—that aims for these two properties:

- *Understandability*: if a type inference system detects a type error, the error should be communicated to the user. The types in existing models have become so complex that they present difficulties even to professional programmers communicating with the type system. This lack of understandability is not acceptable in VPLs aimed at end users.
- *Power without the addition of explicit declarations*: in implicitly typed languages, the introduction of inheritance or prototyping has typically re-introduced explicit declarations of type constraints.

All three properties are of some relevance to textual programming languages, but they are critical in VPLs, because their absence effectively prevents the effective use of static type inference both in VPLs incorporating similarity inheritance and in VPLs aimed at end users.

In this paper, we present a model of types with these three properties in mind. Its purpose is to give early feedback about type safety. 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 flexibly support both coarse- and fine-grained approaches to inheritance, including single inheritance, multiple inheritance, and similarity inheritance.

1.1. Organization of this Paper

After surveying related work in Section 2, we introduce a subset of Forms/3 for formal reasoning purposes in Section 3. (Although the subset is simpler than full Forms/3, it is functionally equivalent to full Forms/3.) Building upon this subset, we present the basic model of types in Section 4, and then expand the model to handle inheritance in Section 5. In Section 6, we prove that the model has the property of soundness with regard to type safety, and discuss the type vocabulary required for communication with users. Finally, after a short discussion of future work in Section 7, we conclude in Section 8.

2. Background and Related Work

2.1. Basic Concepts of Static Polymorphic Type Inference

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 [2] and Schwartzbach [3].

The term *polymorphic types* refers to 'data or programs which have many types or operate on many types' [4]. For a programming language that requires explicit type declarations, a polymorphic variable X may be declared:

$$\text{var } X : \alpha$$

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, and this paper confines its attention to approaches that are entirely static. The primary 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 [5]):

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

The function sumfunc sums the result of applying function *f* to every element of a list, and has type $(\alpha \rightarrow \text{int}) \rightarrow \alpha$ list $\rightarrow$ int, where α is a polymorphic type, and the $\rightarrow$s separate the arguments and the return value in a function's type (e.g., $\alpha \rightarrow$ int means the function takes a polymorphic type α and returns an integer). Suppose three additional functions, cardinal, realfn, and sq, are defined with the following types:

cardinal: int $\rightarrow$ string
realfn: int $\rightarrow$ real
sq: int $\rightarrow$ int

If cardinal or realfn were passed into sumfunc as the first argument (f), then a type error would occur because the system could not resolve the type conflict between string and int or between real and int. If sq is used, however, no type error would arise, and the following inferences could be made:

sumfunc: $(\alpha \rightarrow \text{int}) \rightarrow \alpha$ list $\rightarrow$ int	by initial definition
sq: int $\rightarrow$ int	by initial definition
sumfunc: $(\alpha \rightarrow \text{int}) \rightarrow \alpha$ list $\rightarrow$ int $\Rightarrow \alpha =$ int	by substituting sq for f
sumfunc: (int $\rightarrow$ int) $\rightarrow$ int list $\rightarrow$ int	final result for this use of sumfunc

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

The sumfunc function is an example of *operation polymorphism* [6] because it is code that works on different types that are not necessarily related to one another. *Parametric polymorphism* [2] is a special type of operation polymorphism that works on an infinite number of types, as is true of the sumfunc function example, but the (more general) operation polymorphism does not itself necessarily imply an infinite number of types. Both are 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 multiple related types. We point out this distinction because, unlike other inheritance-supporting languages, our model's inheritance-related polymorphism builds upon operation polymorphism rather than upon inclusion polymorphism.

2.2. Type Inference in Textual Programming Languages

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

2.2.1. Static Types in the Presence of Inheritance

Two well-known languages representing the class-based approach and the prototype-based approach, respectively, are Smalltalk [7] and Self [8]. 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 [9]. 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.* [10] is to derive and solve sets of type constraints. Both of these approaches handle types on a coarse-grained level, namely at the granularity of classes or prototypes.

Imposing a static view of types on a language with inheritance sometimes leads to problematic theoretical issues. These issues arise from the fact that a fundamental difference exists between subtypes and subclasses [11–13]. Subtypes reflect the property of substitutability; they should be able to replace supertypes without introducing type errors [14]. This definition of subtypes allows substitutability of subtypes for supertypes but does not allow overriding in order to specialize a subtype. Subclasses, on the other hand, do allow overriding, because they are simply an implementation convenience for reusing code and do not inherently guarantee anything about substitutability. The difficulty of combining substitutability with overriding in a type system comes from the difficulty of typing methods whose arguments and return type vary from supertype to subtype.

The solution is to separate the notions of subclass and subtype. In separating these two concepts, the problem of covariance versus contravariance becomes clear. *Covariance* typifies the conventional use of inheritance for reuse; method arguments and results in a subclass are allowed to be subtypes of the arguments and results of the class methods. On the other hand, subtyping requires method arguments of a subclass's methods to be *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' [3].

Schwartzbach summarizes a variety of proposed solutions to this dilemma, some of which include: supporting only covariance despite sacrificing type safety, as in Eiffel [15]; 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++ [16]; 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 [17] is an approach that can avoid this problem. It 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 [2], which did not allow inheritance apart from subtyping. Bruce's *bounded matching* [18, 19] 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, Object-Type$\{m_1 : \tau_1; \ldots ; m_k : \tau_k\}$ and ObjectType$\{m_1 : \tau_1; \ \ldots ; m_n : \tau_n\}$, 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 [20] has both types and type classes, and this combination provides some inheritance-like characteristics at a finer granularity than traditional classes. Type classes are declarations of a type's interface and can also include default implementations of interface methods. A type class can inherit interface specifications and default methods from other type classes. A type must implement (or use a default implementation, if one exists) every method in every type class to which it belongs. While most of Haskell's type system allows implicit types which are resolved automatically through unification, explicit declarations are needed of type classes and of user-defined types' membership in them.

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

2.2.2. *Understandability of Type Inference Results*

Although the theoretical foundation of implicit polymorphism is rooted in combinatory logic, Milner [21] was the first to apply the theory to programming languages. Milner's work has been implemented in many textual languages, especially in functional languages (such as Haskell, ML and Miranda) in order to preserve type safety in implicitly typed programs. Within this context, there has been some work in the functional language community related directly to the understandability of type inference results. For Milner-based type inference systems, which reason primarily about functions, understandability of the types is a well-known problem. One reason for this problem is that when higher-order functions are present, types may grow exponentially with respect to the size of the program, as demonstrated in Figures 1 and 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 an erroneous first-order

```
fun pair x y = fn z =>z x y;
let val xl = fn y => pair y y in
                    let val x2 = fn y => xl (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 1. A small program [3]

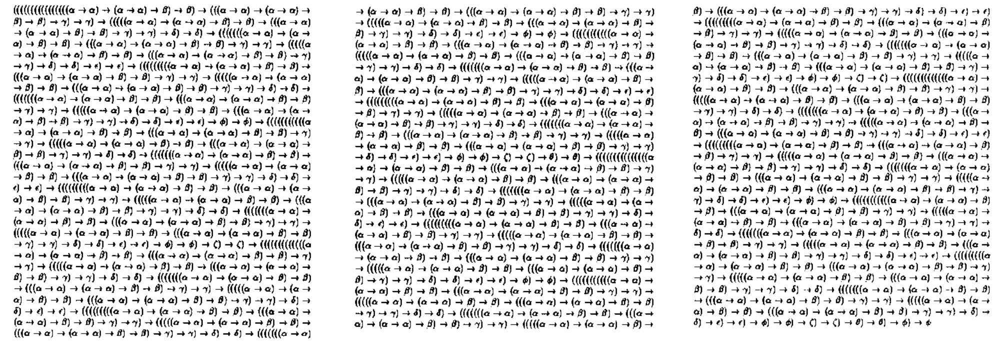


Figure 2. Type of function 'pair' in Figure 1 [3]

program will not type check—can be barriers to the acceptance of type inference systems.

To address this problem, Wand presented an algorithm to isolate and explain type errors [22]. 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 [23]. Their algorithm adds the ability to handle aliased type variables, but it does not handle Haskell's type classes.

Jun and Michaelson [5] presented an approach to improve the ease with which type errors can be recognized, by encoding types with colors. This color visualization approach has been implemented in a visual environment for a subset of standard ML. Each function type is represented as a rectangular block with colored blocks inside that represent argument and result types. Returning to the sumfunc example of Section 2.1, the sumfunc, realfn, and sq functions and the data types real and int can be depicted as in Figure 3. A visual comparison of the blocks shows that the function realfn cannot be substituted into sumfunc because its functional block type scheme conflicts

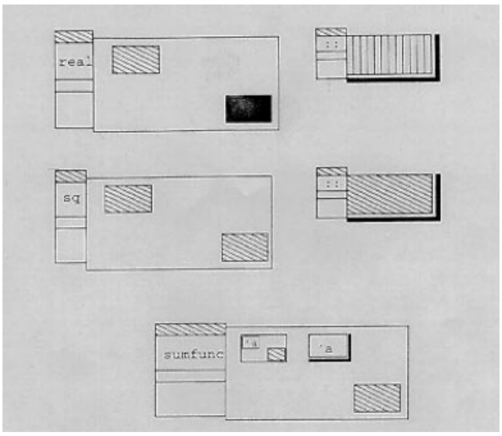


Figure 3. Type visualization for the sumfunc example, from Jun and Michaelson [5]. In the paper describing the approach, colors were replaced by these black and white line patterns. (The label 'real' is not a type, but rather a new name for the function named 'realfn' in the example.) The black block represents a real type, and a diagonally striped block represents an int

with sumfunc's first argument block type scheme. sq's functional block type scheme, however, visually appears to and does fit sumfunc's first argument type scheme. 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.

2.3. Type Systems in VPLs

We have already pointed out that few VPLs use explicit type declarations and that in the absence of explicit type declarations, language designers are left with the choice of either dynamic typing or static typing with type inference. To date, most VPLs (e.g., Prograph [24], KidSim/Cocoa [25], Chimera [26], VIPR [27], and Formulate [28]) have chosen dynamic typing.

Interestingly, the disadvantage of dynamic typing's inability to provide feedback about type errors until run time bears re-examination for responsive VPLs. For responsive VPLs (those languages at liveness level 3 and above on Tanimoto's liveness scale[a]), dynamic type checking can indeed produce immediate feedback about type errors in many cases, due to the fact that at level 3 and above, 'run-time', 'translation

[a]At liveness level 1 no semantic feedback is available. At level 2, the user can obtain semantic feedback, but it is not provided automatically (as in compilers and interpreters). At level 3, incremental semantic feedback is automatically provided after each program edit, and all affected on-screen values are automatically redisplayed (as in the automatic recalculation feature of spreadsheets). At level 4, the system responds to edits as in level 3, as well as to other events such as system clock ticks [29].

time', and 'program-entry time' are intertwined. For example, in spreadsheet languages, which are at level 3, concrete, immediate feedback about type errors can be provided by eagerly evaluating a formula as soon as it is entered, which is even earlier than the feedback about type errors in static approaches for traditional textual languages. If any type error occurs in the course of this evaluation, a special value such as 'Error' is displayed in the cell. This approach features simplicity and immediate visual feedback, but unfortunately, it cannot detect all type errors. For example, if cell A had the value 'true', the type error in the formula 'if cell A then (3 + 4) else (cell A + 4)' would not be detected.

2.3.1. Static Types in VPLs

Our search through VPL literature has revealed only seven VPLs that have incorporated static type inference. In about half of these VPLs, systems like Milner's are fully incorporated into the VPL, and hence static soundness is preserved. ESTL [30] and CUBE [31] 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 [32], has a feature termed *consistency*, with which values can be compared, conditions tested, etc. If such conditions are not met, an inconsistency is said to exist. In this case, the inconsistent area is rendered in a different pattern, and processing of affected areas cease to produce output. This 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 [33, 34] 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 allowed between a function's types supported 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 is supported, no explicit declarations are required, and new user-defined type classes cannot be added.

Clover [35] 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.

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

2.3.2. Understandability of Type Inference Results in VPLs

The remaining VPLs with type inference systems have aimed for greater understandability of type systems, primarily by emphasizing concreteness in the types themselves. Fabrik, [36] which was the first VPL to report the use of type inference, is an example. Fabrik is a dataflow VPL that includes an interactive polymorphic type system with some type inference. Fabrik's type system is simple, concrete and highly visible. Each node in the dataflow graph contains input and output 'pins'. Wires that connect nodes are attached to these pins. Each pin has a type that may be either a primitive type, a compound type constructed from only primitive types or an unspecified (i.e. polymorphic) type. These types can be declared by the user explicitly, or they can be derived implicitly. Type checking is performed when a user attempts to connect two pins. A pin with an unspecified type acquires a type when it is attached to a pin with a known type. If a type mismatch occurs, a message is displayed, and the connection is not made. This approach to implicit polymorphism seems consistent with the concreteness of the language, but the type system is not as fully developed as that of the other languages discussed here. For example, user-defined types are not handled.

In an unusual application of type inference in VPLs, Pacull introduces a visual type system whose goal is not type safety; rather the system infers and propagates information for rendering purposes [37]. 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 [38]. 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.

3. Forms/3 and Core Forms/3

Forms/3 [39] is a research VPL in the spreadsheet paradigm. Our model of types was developed to support not only features of Forms/3 such as cells, matrices (spreadsheet-like grids of cells), and graphical types, that are also found (or emerging) in other spreadsheet-like languages, but also advanced features of Forms/3 such as similarity inheritance. In this section, we first informally present an example from Forms/3 to paint a concrete picture of a setting in which the type system is expected to function. We then formally define 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.

3.1. An Informal Introduction to Forms/3

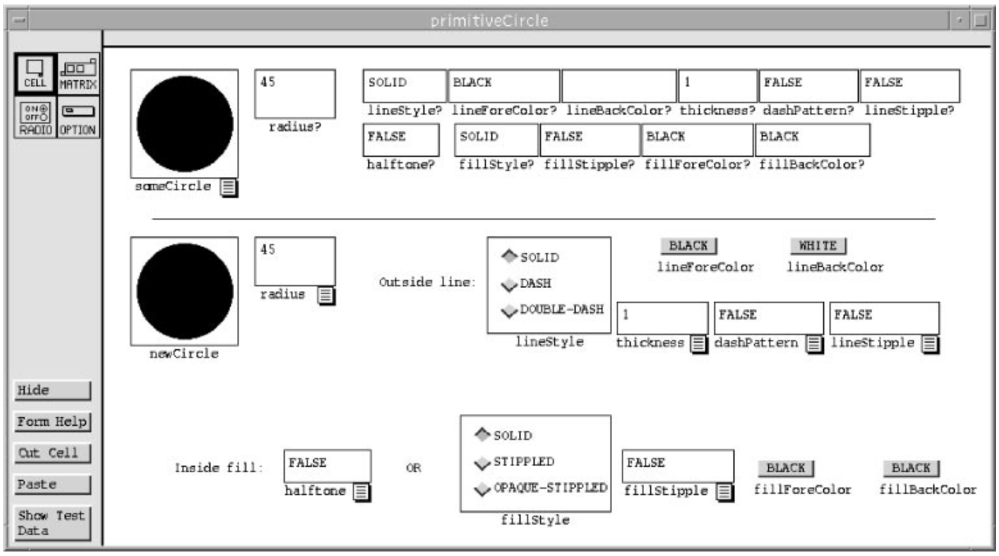
Forms/3 programs include forms (spreadsheets) with cells, but the cells are not locked into a grid. A Forms/3 programmer creates a program by using direct manipulation to place cells on forms and defines a formula for each cell using a flexible combination of pointing, typing, and gesturing. A program's calculations are entirely determined by these formulas. As in other spreadsheet languages, Forms/3 formulas can be thought of as a network of (one-way) constraints, and the system continuously ensures that all values displayed on the screen satisfy these constraints.

Forms/3 supports both built-in types and user-defined types. A type is described by a type definition form. Attributes of a type are defined by formulas in groups of cells on that form, and instance(s) of a type are instantiated by one or more cells on that form that can be referenced in formulas just like any other cell. Built-in types are provided in the language implementation but are otherwise identical to user-defined types. For example, the circle shown in cell newCircle in Figure 4 is defined by cells specifying its radius, line thickness, color, and other attributes.

Suppose a spreadsheet user such as a population analyst would like to define a visual representation of data using domain-specific visualization rules that make use of the built-in circle type of Figure 4. Figure 5(a) shows such a visualization in Forms/3. The program categorizes population data into cities, towns, and villages, and represents each with a differently sized black circle. Each of these different black circles is defined by a different copy of the form in Figure 4.

Users can specify formulas either textually or graphically. For example, the population analyst can define the size of a circle by entering a spreadsheet formula in the conventional textual manner, such as by typing a formula into the radius cell of Figure 4. Alternatively, users can define formulas via direct manipulation and gestures, entering a circle-shaped gesture in the formula edit window for a cell intended to result in a circle, instead of typing a formula into the radius cell. For example, to define the formula for cell city, the population analyst draws the circle gesture in Figure 5(b). This is syntactic sugar for the mechanism of Figure 4: it defines the cell's formula to be a reference to cell newCircle on a copy of the built-in circle definition form whose radius formula is defined to be the radius of the drawn circle gesture. To specify that the circle should be black, the population analyst clicks on the circle to display its definition form, and then defines the desired formula for cell fillForeColor [Figure 5(c)]. (Full details on programming in Forms/3 by direct manipulation and gestures are given in Burnett and Gottfried [39].)

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 [40, 41]. However, because our approach to types assumes homogeneous streams, reasoning about streams



someCircle
radius? 45
lineStyle? SOLID
lineForeColor? BLACK
lineBackColor?
thickness? 1
dashPattern? FALSE
lineStipple? FALSE
halftone? FALSE
fillStyle? SOLID
fillStipple? FALSE
fillForeColor? BLACK
fillBackColor? BLACK

newCircle
radius 45
Outside line: lineStyle: SOLID / DASH / DOUBLE-DASH
lineForeColor BLACK
lineBackColor WHITE
thickness 1
dashPattern FALSE
lineStipple FALSE
Inside fill: halftone FALSE OR fillStyle: SOLID / STIPPLED / OPAQUE-STIPPLED
fillStipple FALSE
fillForeColor BLACK
fillBackColor BLACK

Figure 4. A portion of a Forms/3 form (spreadsheet) primitiveCircle, which defines a circle. (The form name can be seen in the window title bar.) A user can view and specify spreadsheet formulas by clicking on the formula tabs (▤). Radio buttons and popup menus are the equivalent of cells with simple formulas. Above the line, the circle in someCircle is a sample circle (or the user can edit its formula to refer to some other circle), and all the other cells above the line report on its attributes. Below the line, the circle in newCircle is constructed using the specifications in the other cells below the line

can be done as a unit, in the same manner as if each formula did simply define an atomic value. Hence, the only result of considering time in our approach to types would be to complicate the notation, and we will thus ignore it in this paper.

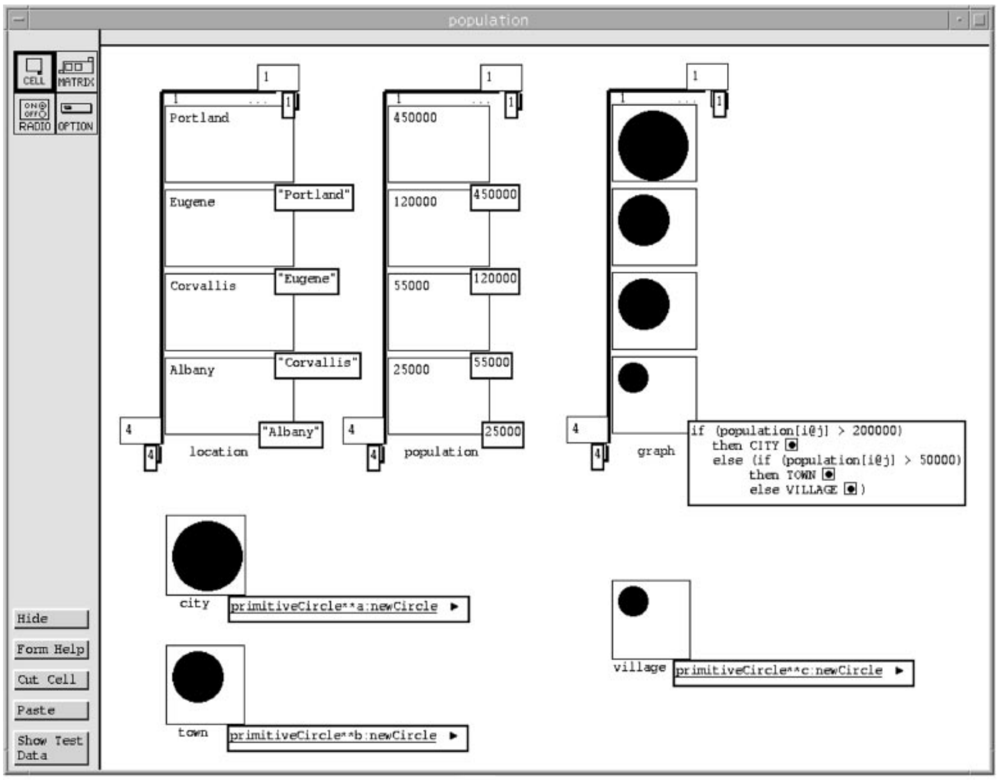
Forms/3 has recently incorporated a fine-grained approach to inheritance termed ‘similarity inheritance’. We will consider this feature and its impact on type inference in Section 5.

3.2. Core Forms/3: The Subset for Formal Reasoning

3.2.1. Programming Objects and Notational Conventions

In the previous section, the programming objects in Forms/3 were informally used as follows. A program consisted of a set of forms, such as the population and primitiveCircle forms, each of which contained cell groups and cells, which in turn had formulas. In this section, we revisit these programming objects formally in Core Forms/3, a subset of Forms/3 for formal reasoning purposes. The following define the basic programming objects in Core Forms/3:

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



location	population
Portland	450000
Eugene	120000
Corvallis	55000
Albany	25000

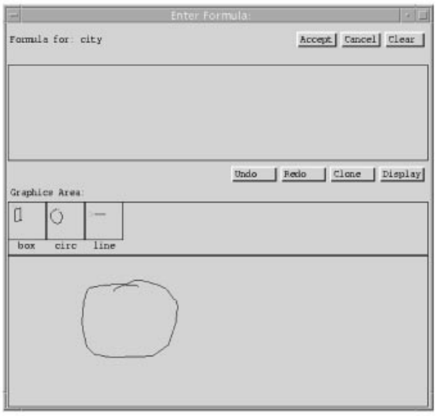
```
if (population[i@j] > 200000)
   then CITY ●
   else (if (population[i@j] > 50000)
        then TOWN ●
        else VILLAGE ●)
```

city: primitiveCircle**a:newCircle

town: primitiveCircle**b:newCircle

village: primitiveCircle**c:newCircle

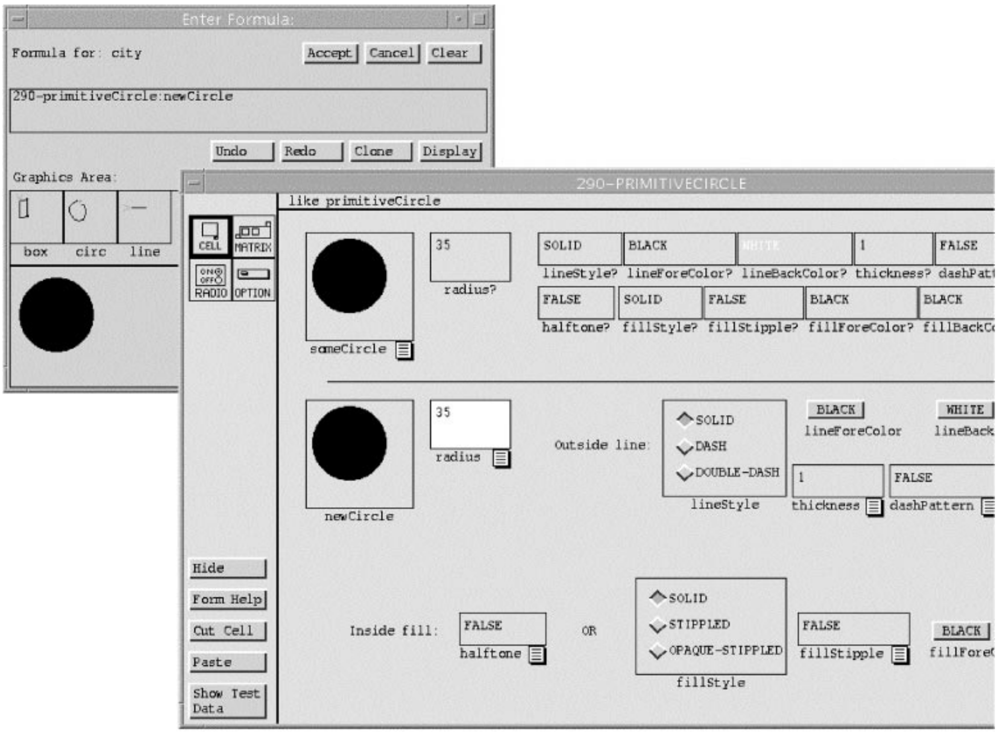
(a)



Formula for: city

(b)

Figure 5. (a) A visualization of population data. The formula shown for graph is shared by the four cells inside the matrix. (The ● s in the formula are miniaturized drawings of the cells' current values, which can optionally be included in formula displays.) (b) To define the circle for cell city, the population analyst first draws a circle gesture in city's formula edit window, and then, (c) after clicking on the resulting circle to display its definition form (in gray because it is a copy; white indicates formulas different from the original), the population analyst specifies the black fillForeColor formula via a popup menu. Each manipulation is immediately reflected textually and graphically in city's formula edit window (shown behind the circle form)



Formula for: city

290-primitiveCircle:newCircle

like primitiveCircle

sameCircle

radius? 35

lineStyle? SOLID lineForeColor? BLACK lineBackColor? WHITE thickness? 1 dashPatt FALSE

halftone? FALSE fillStyle? SOLID fillStipple? FALSE fillForeColor? BLACK fillBackCo BLACK

newCircle

radius 35

Outside line: lineStyle ◆SOLID ◇DASH ◇DOUBLE-DASH

lineForeColor BLACK lineBack WHITE

thickness 1 dashPattern FALSE

Inside fill: halftone FALSE OR fillStyle ◆SOLID ◇STIPPLED ◇OPAQUE-STIPPLED

fillStipple FALSE fillFore BLACK

(c)

Figure 5. *(Continued)*

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

$$\text{modelName} = \begin{cases} \text{formID if this form is not a copy or is a promoted copy} \\ \quad \text{(defined in Section 5)} \\ \text{F.modelName if this form is a copy of form F} \end{cases}$$

- An *ROset* is a set of referenceable objects, each of which is identified by a unique cellID.
- A *referenceable object* (RO) is a cell or a cell group.
- A *cell group* is a matrix or an abstraction box.
- A *cell* is a tuple (cellID, ROset, formula, value) whose ROset contains only (zero or more) virtual ROs (defined later in this section).
- 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 a matrix's ROset—{⟨MID⟩[numrows], ⟨MID⟩[numcols]}.)
- A *formula* is as defined in Table 1.

The reason for the modelName element of forms is to explicitly track similarities among forms, which will allow some inferences about a form's types to be made based upon the form's similarities to another form. The intuition behind ROs is that a referenceable object is any non-constant that can be referenced in a formula. But 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'. Note from these definitions that it is not possible in Core Forms/3 for a cell or cell group to exist without a formula. 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.

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. Figure 5 shows examples of each of these programming objects as they exist in full Forms/3. The program consists of the population form and several copies of the primitiveCircle form. For Figure 5(a), the form id and its modelName are both 'population'. Form population's ROset consists of cells city, town, and village, and of matrices location, population, and graph.

In addition to the above objects, *Visual Abstract Data Type* (VADT) forms are type definition forms that support both built-in and user-defined types. The primitiveCircle form of Figure 4 and the Point form of Figure 6 are examples of VADT forms in full Forms/3. VADT forms and their components are defined as follows:

- A *VADT form* is a form whose ROset includes a cell with cellID 'Image', one abstraction box with cellID 'MainAbs', and zero or more additional ROs.
- 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 a cell's ROset.

The purpose of a VADT form whose formID is T is to define a type named T. Both built-in and user-defined types, including recursive types, are defined by VADT forms [39]. An abstraction box on the VADT form defines the group of ROs from which an instance of T is constructed, and hence the value of each abstraction box on a form with formID T is an instance of type T. For example, in Figure 4, the abstraction boxes are newCircle and someCircle, and each has as its value an instance of type primitiveCircle. (For built-in types such as circles, the component ROs of the abstraction box that would reveal the implementation of the type are not visible on the screen.) Cell names are actually cosmetic attributes to help document functionality, and in this paper often correspond to the behind-the-scenes cellIDs; however, in Figure 4, the actual cellID for someCircle is 'MainAbs'. The formula for cell Image on VADT form T defines the appearance of an instance of type T, and Image's value shows the formula applied to the instance of T defined in MainAbs. Figure 4's cell Image was right-truncated from the screen shot. In Figure 6, cell Image is the upper right cell, and the abstraction boxes

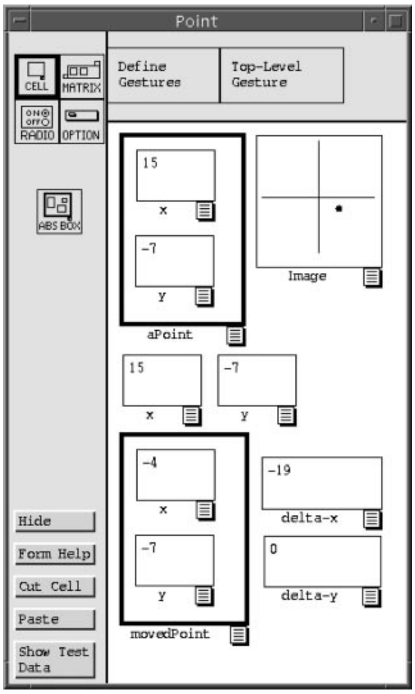


Figure 6. The Point VADT form defines type Point and two of its instances: Each of the abstraction boxes aPoint and movedPoint has a value of type Point

are aPoint and movedPoint; aPoint is the one whose cellID behind the scenes is 'MainAbs'. 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 paper 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 ('ref') when the operand is an RO. 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 ('ref') when the operand is a constant. For example, $X \leftarrow_c C$ means that RO X's formula is the constant C.
- $\overset{+}{\in}$ denotes the transitive closure of $\in$. $X \overset{+}{\in} Z$ iff either $X \in Z$, or $X \in Y$ and $Y \overset{+}{\in} Z$.

3.2.2. Formula Syntax and Semantics

As Table 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

Table 1. The grammar for Core Forms/3's formula language. The divider separates the operator syntax from the ref operand syntax. (The third line of the top section is not useful in the grammar, but is included to make clear how the implicit operators relate to the explicit elements of the grammar). To minimize the amount of new notation, we also build upon the terms established here in the formal presentation of the model, as explained in the text

formula	::= *compositionOfParts* | expr
expr	::= *constant* | ref
exprShowingImplicits	::= $\leftarrow_c$ *constant* | $\leftarrow$ ref
ref	::= RORef | formRef : RORef
formRef	::= *formID* | *modelName* defSet
defSet	::= (defs)
defs	::= def | def, defs
def	::= RORef = expr
RORef	::= cellID
cellID	::= *simpleCellID* | *matrixID* | *matrixID* [subscripts] | *absID* | *absID* [*simpleCellID*] | *absID* [*matrixID*] | *absID* [*matrixID*] [subscripts]
subscripts	::= matrixSubscript@matrixSubscript
matrixSubscript	::= expr

('$\leftarrow_c$'), and the explicit operator 'compositionOfParts'. Table 2 defines the semantics of each of these operators in terms of their preconditions and postconditions. In full Forms/3, the cells in matrix graph in the population example [Figure 5(a)] provide examples of the first operator because they reference cells city, town and village. The cells in matrix population provide examples of the second operator, because the formula for each is simply a constant. The abstraction box aPoint in Figure 6 provides an example of the third operator, which is implicit in full Forms/3, because cells x and y are inside aPoint.

Returning to Core Forms/3, note that there are no explicit input operators such as 'read' or 'getc'. As in most spreadsheet languages, inputs are entered into the program via constant formulas. This characteristic of spreadsheet languages provides opportunities not found in many other paradigms regarding the ability to apply (incremental) static type checking to every part of a program.

The syntax given in Table 1 includes textual versions of some elements that the user would never actually type in (in either Core Forms/3 or Forms/3). For example, the user never uses the textual defSet syntax shown; instead he/she clicks on the desired cell and the system internally records the reference using the defSet notation above. However, we present the internal notation here because it is a useful formal notation to express the type reasoning mechanism. Also, for the purposes of this paper, we take advantage of the notation in Table 1 as follows. Unless specifically used in the context of a formula syntax example, we will use ref syntax as an abbreviation for the forms and ROs themselves. For example, 'form F' will be used as an abbreviation for 'the form whose id is F', and 'F : A' will be used as an abbreviation for: 'the RO whose cellID is A that is an element of the form whose formID is F'.

Table 2. Axiomatic semantics for each operator in Core Forms/3. (The special provisions for matrices are primarily because matrices do not have values.) Implicit in the notion of equality is the fact that if value1 = value2, then their types are equal. Also note that the '$Y \overset{+}{\not\leftarrow} X$' precondition prevents circular references. All the preconditions are easily checked statically, and in full Forms/3 are enforced by the environment

Formula for RO X	Preconditions	Postconditions
$\leftarrow Y$, where Y is an RO	Y is an existing cell or abstraction box $Y \overset{+}{\not\leftarrow} X$ X is not a matrix	X.value = Y.value |X.ROset| = |Y.ROset| $\forall Y_i \in$ Y.ROset: $X_i \in$ X.ROset $X_i \leftarrow Y_i$
	Y is an existing matrix $Y \overset{+}{\not\leftarrow} X$ X is a matrix	(matrices do not have values) |X.ROset| = |Y.ROset| $\forall Y_i \in$ Y.ROset: $X_i \in$ X.ROset $X_i \leftarrow Y_i$ X_i.id = X.id[row$_i$@col$_i$] iff Y_i.id = Y.id[row$_i$@col$_i$]
$\leftarrow_c C$, where C is a constant	X is not a matrix	X.value = C
compositionOfParts	X is a cell group	X.value = $\{X_i.\text{value} \mid X_i \in \text{X.ROset}\}$ iff X is not a matrix

3.2.3. 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 referring to ROs on 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. The '+' form in Table 3 consists of a set of three ROs: plusA, plusB and plusC. The cells plusA and plusB are the two arguments for addition. They are modifiable in order to allow passing in different arguments for different additions needed in a program. The cell plusC contains the result and is therefore not modifiable. It is this cell, plusC, that can be referenced in Core Forms/3 to mean the same thing as 'plusA + plusB' in full Forms/3.

Table 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 provide access to a primitive operation; the others are: −, *, /, mod, =, and, or, >, <, not, width, height, if, and compose

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

The defSet notation of Table 1 is generated by the system when copies are made, and enumerates formulas that make a copied form different from the form from which it was copied. For example, the notation if(ifB ≡ 100) indicates a copy of form ‘if’ on which cell ifB contains the constant formula 100. Hence, ‘$\leftarrow +$(plusA ≡ 10, plusB ≡ 100): plusC’ in Core Forms/3 would mean the same as ‘10 + 100’ in full Forms/3. In this paper, we will use the defSet notation generated by the system as a formal notation as well.

As Table 3 implies, all copies of + have the ‘same’ cells on them, although their formulas for plusA and plusB may be different. More precisely (and more generally), this relationship is described as the first property in Table 4, which says that forms with the same modelName always have ROsets whose cells have the same cellIDs.

The rest of the properties given in Table 4 are particular to VADT forms. As we pointed out before, VADT forms can be used to define new types, and all built-in types are also described with VADT forms, such as primitiveCircle in Figure 4. There is also a primitiveNumber form, primitiveBoolean form, etc. (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.) The definitions of Section 3.2.1 distinguish one abstraction box on each VADT form by cellID ‘MainAbs’. 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 (from properties [AbsType] and [AbsStruct]). [AbsType] further says that every abstraction box on VADT form T results in a value of type T. For example, recall that aPoint and movedPoint on form Point in Figure 6 are both of type Point.

What [VADTformExistsC], [VADTformExistsR], and [Inst] say is that, for every RO X, there exists a VADT form denoted T_X whose main abstraction box references X. As a result, T : MainAbs.value = X.value, and X.value is of type T. Further, each T_X 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. For example, if the constant 3 of type primitiveNumber exists, then there exists a form primitiveNumber_3 = primitive Number(MainAbs ≡ 3), whose modelName is primitiveNumber. Similarly, if some cell Z has a Point (15, −7) as its value, then a copy of form Point exists. [Inst] states further that if Z has such a value, it directly or transitively references an abstraction box on the appropriate VADT form (Point, in this example).

Table 4. Invariant properties of forms. Entries such as 'T(MainAbs ≡ C)' follow the defSet syntax established in Table 1. (The subscript notation for set elements is used to represent arbitrary elements in the set, and does not imply any position in the set, since sets are inherently unordered)

[Copies]	$\dfrac{\text{F and G are forms, F.modelName} = \text{G.modelName, } f_i \in \text{F.ROset}}{\lvert\text{F.ROset}\rvert = \lvert\text{G.ROset}\rvert,\ g_i \in \text{G.ROset, } g_i\text{.cellID} = f_i\text{.cellID}}$
[VADTformExistsC]	$\dfrac{\text{C is of type T, where C is a constant}}{\exists \text{ VADT form } T_C, \text{ where } T_C = \text{T(MainAbs} \equiv \text{C}), T_C\text{.modelName} = \text{T}}$
[VADTformExistsR]	$\dfrac{\text{X.value is of type T, where X is an RO}}{\exists \text{ VADT form } T_X, \text{ where } T_X = \text{T(MainAbs} \equiv \text{X}), T_X\text{.modelName} = \text{T}}$
[Inst]	$\dfrac{\text{X.value is of type T, where X is a cell and X} \leftarrow \text{someRO}}{\text{X} \overset{+}{\leftarrow} \text{T:Y, where Y is an abstraction box and T.modelName} = \text{T}}$
[AbsType]	$\dfrac{\text{T:X is an abstraction box}}{\text{X.value is of type } T_X\text{.modelName} = \text{T.modelName}}$
[AbsStruc]	$\dfrac{\text{T:X, T:Y are abstraction boxes, } X_i \in \text{X.ROset}}{\lvert\text{Y.ROset}\rvert = \lvert\text{X.ROset}\rvert,\ Y_i \in \text{Y.ROset, } Y_i \text{ is a matrix iff } X_i \text{ is a matrix}}$

3.3. 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 in Figure 4). 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 which are 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 [39]), 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

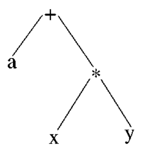


Figure 7. Translation of the Forms/3 formula (a + (x * y)) to Core Forms/3 proceeds bottom-up, first translating (x * y) to *(timesA ≡ x, timesB ≡ y): timesC, and then incorporating that translation into the full expression, resulting in the Core Forms/3 formula being a reference to + (plusA ≡ a, plusB ≡ *(timesA ≡ x, timesB ≡ y): timesC): plusC

an expr (following Table 1). If the prefix version of X.formula is op $expr_1, expr_2, \ldots$, then X.formula in Core Forms/3 is the following reference:

$$op(arg_1 \equiv expr_1, arg_2 \equiv expr_2, \ldots)\text{: Result}$$

where $arg_1, arg_2, \ldots$ are the cellIDs of modifiable ROs on form op, and Result is the cellID of an unmodifiable RO containing the result of the built-in operation. (The notation is that defined for defSets in Table 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 7 for an example.

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.

4. Model of Types

4.1. Fine-grained Reasoning in Terms of Guarantees versus Requirements

The goals of this research were to develop a model of types comprehensive enough to be usable in VPLs for programmers with powerful features such as polymorphism and fine-grained inheritance, while at the same time being potentially understandable enough for use even in VPLs intended for end users. (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 fine-grained level about individual operations guaranteed and required rather than about entire types as atomic units.

At this granularity, reasoning about the operations that can be guaranteed compared to those required 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.

For example, if cell X is being added to something, then a requirement exists that operation + (actually, +:plusC in Core Forms/3) be supported for X. If cell X is able to be added to something, then operation + is guaranteed for X. If all the requirements are met by the guarantees, the program is type safe.

More formally, if the set of operations that are inferred by our model to be guaranteed for RO X are denoted G(X), and the set of operations inferred to be required of RO X are denoted R(X), then in this model of types, type safety is defined as follows:

Definition. If ∀ROs X $\overset{+}{\in}$ program P, R(X) ⊆ G(X), then P is *type safe.*

Because all reasoning is done in terms of guarantees and requirements in our model, there are no subtypes, complex compositions of types, or interfaces. Instead, as the above definition implies, if the requirements are not a subset of the guarantees, then there is a type error.

The model of types presented here is intended to be applied incrementally, doing as much analysis or reanalysis as is necessary after each user edit. (Recall from the discussion in Section 2.3 that—even given incremental execution—due to laziness or short-circuit evaluation, static type inference can detect errors that go undetected under dynamic typing.) Given such an incremental implementation of the model, the understandability potential of error messages is greater than in a non-incremental implementation, because if the program was previously type safe, then any type error incrementally reported must have been introduced by the just-entered program edit. For Forms/3, this is the formula just entered.

The next two sections describe how the guarantees and requirements are statically inferred by our type system.

4.2. Guarantee Sets

In general, each RO guarantees all the operations defined on the VADT form corresponding to its value (this form was defined by [VADTformExistsR] in Table 4). In Core Forms/3, each operation is associated with an RO.

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

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

This axiom is applicable to both user-defined types and built-in types. For example, G(primitiveCircle:newCircle) includes all the ROs on form primitiveCircle: the operators radius, thickness, lineStyle, lineForeColor, etc., including the abstraction boxes someCircle and newCircle (refer back to Figure 4 to view primitiveCircle). Other circle-related tasks that can be constructed using these low-level operations do not need to be included on the primitiveCircle form itself. In this paper, we abbreviate the set of low-level operations for these primitive types (the ROs on their VADT forms) as '⟨primitiveType⟩Operations'; in this example 'primitiveCircleOperations'.

Table 5. Every constant value guarantees exactly the operations on its primitive form. For succinctness, we abbreviate these sets ⟨primitiveType⟩Operations rather than listing the individual ROs

Example constants C	G(C)
C = 3	primitiveNumberOperations
C = 'hello'	primitiveTextOperations
C = true	primitiveBooleanOperations

Using the notation established in Tables 1 and 4, axiom [GC] says that ROs with constant formulas simply derive their guarantee sets from the primitive form describing the constant's value (see Table 5):

[GC] $$\frac{X \leftarrow_c C}{G(X) = G(C)} \quad \begin{array}{l}\text{where } C \text{ is a constant}\\ \text{where } G(C) = \{y \mid y \overset{+}{\in} F_C.\text{ROset}\}\end{array}$$

For cells and matrices referencing other cells and matrices, the guarantee set simply propagates from the referenced cell or matrix by axiom [Gref] below. The 'where' clause restricts this axiom to cells and matrices 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] $$\frac{X \leftarrow Y}{G(X) = G(Y)} \quad \text{where } X \text{ is a cell or matrix}$$

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

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

[GM] is one of several elements in this model that are different for matrices than for other ROs. 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 shows 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.

Table 6. Guarantee sets for some primitive forms. Type postconditions are stated as guarantees under our model of types; this table shows these postconditions for a few of the primitive forms. (Compare these with the previous type-related postconditions of Table 3.) Preconditions for these forms will be given in the next subsection

ModelName	ROset	Type-related postconditions (Guarantees)
+	Contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC	G(plusC) = primitiveNumberOperations
>	Contains modifiable cells with cellIDs greaterthanA, greaterthanB, and unmodifiable cell with cellID greaterthanC	G(greaterthanC) = primitiveBoolean Operations
append	Contains modifiable matrices with cellIDs appA, appB, and unmodifiable matrix with cellID appC	$G(appC) = G(appA) \cap G(appB)$
if	Contains modifiable cells with cellIDs ifA, ifB, ifC, and unmodifiable cell with cellID ifD	$G(ifD) = G(ifB) \cap G(ifC)$

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 6.

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.

From the previous paragraph, the requirements axiom needed might seem to be that the requirements of an RO are the union of all the requirements of ROs referencing it, as in

$$\frac{X_1,\ X_2, \ldots, X_n \leftarrow Y}{R(Y) = \cup R(X_i)}$$

for $i = 1 \ldots n$, where X_i is not an abstraction box.

Abstraction boxes referencing Y are already ‘safe’ from type errors because their definitions and properties prevent impersonations: assignments of invalid types into an abstraction box. Hence, it is not necessary to propagate requirements from abstraction boxes that reference Y into the requirements of Y, and such references can simply be ignored in the requirements axiom. Taking advantage of this fact propagates fewer requirements, which encourages polymorphism.

The above version is adequate for propagating requirements that are already present in the system, but it does not initiate new ones. The primary way for new requirements to be initiated is for an operation on a VADT form to be referenced. That is, R(Y) needs to include not only the operations given in the axiom version above, but also those operations in F_Y.ROset that are actually being used (referenced). Adding an appropriate precondition and union factor to address this need leads to a version something like the one below (the new additions are on the right):

$$\frac{X_1,\ X_2, \ldots, X_n \leftarrow Y \quad \text{and} \quad Z_1 \leftarrow F_Y : Op_1,\ Z_2 \leftarrow F_Y : Op_2, \ldots, Z_m \leftarrow F_Y : Op_m}{R(Y) = \cup\, R(X_i) \quad \cup \quad \{Op_1,\ Op_2, \ldots Op_m\}}$$

for $i = 1 \ldots n$, where X_i is not an abstraction box.

However, this version overlooks the fact that copies of F_Y (multiple VADT forms whose main abstraction boxes reference Y) may have individual differences in some of the formulas of their other ROs. For example, there could be two copies of form Point (from Figure 6) with aPoint on both copies referencing a cell Y, but with cell delta-x on one of the copies referencing some cell V, and with cell delta-y on the other copy referencing some cell W. In defSet notation, the first copy is Point(aPoint ≡ Y, delta-x ≡ V) and the second copy is Point(aPoint ≡ Y, delta-y ≡ W). Thus, additional requirements for Y are that its type definition form also includes operations that appear in these defSets, namely aPoint, delta-x, and delta-y in this example. Changing the second precondition to defSet notation and adding a third union factor to the conclusion leads to the final version below:

$$[\text{R1}] \quad \frac{X_1,\ X_2, \ldots, X_n \leftarrow Y \quad \text{and} \quad Z_1 \leftarrow F_Y(\text{defSet}_1) : Op_1,\ Z_1 \leftarrow F_Y(\text{defSet}_2) : Op_2, \ldots, Z_m \leftarrow F_Y(\text{defSet}_m) : Op_m}{R(Y) = \cup R(X_i) \quad \cup\ \{Op_1,\ Op_2, \ldots Op_m\} \cup \{Op \mid Op \in \text{arg}_k\} \ \text{where} \ \text{arg}_k = \{\text{arg} \mid \text{arg} \equiv \text{expr} \in \text{defSet}_k\}}$$

for $i = 1 \ldots n$, $k = 1 \ldots m$, where X_i is not an abstraction box.

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

Turning now to matrices, let M be a matrix. M’s numrows and numcols cells are always known to require primitiveNumberOperations:

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

where N ∈ M.ROset and either N.ID = M.ID[numrows] or N.ID = M.ID[numcols].

Table 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 preconditions for a few of the primitive forms. (The preconditions are invariant, but to avoid clutter, we did not explicitly repeat them in the postconditions of Table 6)

ModelName	ROset	Type-related preconditions: Requirements
+	Contains modifiable cells with cellIDs plusA, plusB, and unmodifiable cell with cellID plusC	R(plusA) = primitiveNumber Operations R(plusB) = primitiveNumber Operations
>	Contains modifiable cells with cellIDs greaterthanA, greaterthanB, and unmodifiable cell with cellID greaterthanC	R(greaterthanA) = primitiveNumber Operations R(greaterthanB) = primitiveNumber Operations
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)[a] If tempR = application of [R1] to appB, then R(appB) = tempR∪R(appC)
if	Contains modifiable cells with cellIDs ifA, ifB, ifC, and unmodifiable cell with cellID ifD	R(ifA) = primitiveBooleanOperations If tempR = application of [R1] to ifB, then R(ifB) = tempR∪R(ifD) If tempR = application of [R1] to ifC, then R(ifC) = tempR∪R(ifD)

[a] Or, more formally,

$$\frac{X_1,\ X_2, \ldots X_n \leftarrow \text{appA} \ \text{ and } \ Z_1 \leftarrow F_{\text{appA}}\ (\text{defSet}_1) : \text{Op}_1,\ Z_2 \leftarrow F_{\text{appA}}\ (\text{defSet}_2) : \text{Op}_2, \ldots\ Z_m \leftarrow F_{\text{appA}}\ (\text{defSet}_m) : \text{Op}_m}{R(\text{appA}) = \cup\ R(X_i) \cup \{\text{Op}_1,\ \text{Op}_2, \ldots\ \text{Op}_m\} \cup R(\text{appC})}$$

for $i = 1 \ldots n$.

Matrices require everything that the cells in their gridROsets require:

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

Regarding primitive forms, preconditions on primitive forms' ROs add to the requirements propagating through the system (see Table 7). For example, in the + form, the ROs plusA and plusB now require primitiveNumberOperations in addition to the preconditions previously given for form +.

4.4. Recursion

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

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

$$\text{[RecC]}\quad \frac{\text{F}:\text{X} \leftarrow \text{if(ifA} \equiv \text{Adef, ifB} \equiv \text{Bdef, ifC} \equiv \text{Cdef}):\text{ifD, there is a recursive reference in Cdef}}{\text{G}(\text{F}:\text{X}) = \text{G}(\text{if(ifA} \equiv \text{Adef, ifB} \equiv \text{Bdef, ifC} \equiv \text{Cdef}):\text{ifB})}$$

Obviously, if there are nested 'if's involved, both branches could be recursive. This does not cause difficulties, since the axiom can simply be applied recursively. However, it is also possible to construct multi-recursive 'if's that never terminate at all because there is no base condition provided, and the axiom set presented here does not provide a way to reason about the types of such non-terminating programs.

4.5. Example: Type Inference (Without Inheritance)

The population example in Figure 5 is a program with no inheritance. The population program provides different-sized circles to represent cities depending on their population. For this type inference example, we use full Forms/3 because it is a real environment and thus allows screenshots of example programs. 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 3. (For conciseness, we omit form names where doing so does not introduce ambiguity.)

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

$$\text{R}(\text{location[1@1]}) = \{\} \qquad \text{[R1]}$$
$$\text{G}(\text{location[1@1]}) = \text{G}(\text{'Portland'}) = \text{primitiveTextOperations} \qquad \text{[GC]}$$

Obviously, $\text{R}(\text{location[1@1]}) \subseteq \text{G}(\text{location[1@1]})$, 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. For precise reasoning, we can translate the formula for the cells in the graph matrix's gridROSet to the Core Forms/3 equivalent; for example, there is a copy of the if form

[b]Some languages' type inference systems require a special construct for recursive calls to allow static detection of recursion. In Core Forms/3, recursion can be detected statically without such a device. There is a recursive reference in Bdef if either $\text{F}':\text{X}$ is referenced in Bdef, or else there is a recursive reference in $\{\text{recdef}_1 \mid \text{Bdef} = \text{someForm}(\text{arg}_1 \equiv \text{recdef}_1,\ \text{arg}_1 \equiv \text{recdef}_2, \ldots):\text{someCell}\}$, where F.modelName $=$ F$'$.modelName.

on which ifB ← population : city. In the context of that copy of the 'if' form, the reasoning proceeds as follows:

$$\begin{aligned}
R(\text{city}) &= R(\text{ifB}) && [\text{R1}]\\
&= R(\text{ifD}) && [\text{Table 7}]\\
&= R(\text{graph}[1@1]) \cup R(\text{graph}[2@1]) \cup R(\text{graph}[3@1]) \cup R(\text{graph}[4@1]) && [\text{R1}]\\
&= \{\,\} && [\text{R1}]
\end{aligned}$$

However, for a more direct translation to the (real) programs displayed in full Forms/3 screenshots, we will present the implications of these precise steps to the full Forms/3 programs. Thus, the equivalent of the above in the full Forms/3 program is:

$$\begin{aligned}
R(\text{city}) &= R(\text{'then' part of graph's 'if' formula}) && [\text{R1}]\\
&= R(\text{result of graph's 'if' formula}) && [\text{Table 7}]\\
&= R(\text{graph}[1@1]) \cup R(\text{graph}[2@1]) \cup R(\text{graph}[3@1]) \cup R(\text{graph}[4@1]) && [\text{R1}]\\
&= \{\,\} && [\text{R1}]
\end{aligned}$$

Continuing from here (the same way in Core Forms/3 as in full Forms/3) finishes up the reasoning about the requirements on cell city:

$$G(\text{city}) = G(\text{290-primitiveCircle : newCircle}) = \text{primitiveCircleOperations} \qquad [\text{GC}]$$
$$\text{Thus, } R(\text{city}) = \{\,\} \subseteq \text{primitiveCircleOperations} = G(\text{city})$$

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

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

$$\begin{aligned}
&R(\text{graph}[1@1]) = \{\,\} && [\text{R1}]\\
&G(\text{graph}[1@1]) = G(\text{if1 : ifD}) \quad \text{i.e., result of graph's 'if' formula} && [\text{Gref}]\\
&\quad = G(\text{city}) \cap G(\text{if2 : ifD}) \quad \text{i.e. } G(\text{city}) \cap \text{result of graph's inner 'if' subexpression} && [\text{Table 6}]\\
&\qquad = G(\text{city}) \cap (G(\text{town}) \cap G(\text{village})) && [\text{Table 6}]\\
&\qquad = \text{primitiveCircleOperations} && [\text{GC}]\\
&R(\text{graph}[1@1]) = \{\,\} && [\text{R1}]\\
&\text{Thus, } R(\text{graph}[1@1]) = \{\,\} \subseteq \text{primitiveCircleOperations} = G(\text{graph}[1@1])
\end{aligned}$$

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 primitiveNumberOperations. (Forms ‘ > 1’ and ‘ > 2’ are copies of form ‘ >’ on which greaterthanA ← population[1@1].)

$$\begin{aligned}
\mathrm{R}(\text{population}[1@1]) &= \mathrm{R}(>1:\text{greaterthanA}) \cup \mathrm{R}(>2:\text{greaterthanA}) \\
&\quad \text{i.e. the union of the requirements on the first} \\
&\quad \text{arguments of both ‘ > ’ comparisons} && [\text{R1}] \\
&= \text{primitiveNumberOperations} && [\text{Table 7}] \\
\mathrm{G}(\text{population}[1@1]) &= \mathrm{G}(450000) && [\text{GC}] \\
&= \text{primitiveNumberOperations}
\end{aligned}$$

$$\begin{aligned}
\text{Thus, } \mathrm{R}(\text{population}[1@1]) &= \text{primitiveNumberOperations} \\
&\subseteq \text{primitiveNumberOperations} \\
&= \mathrm{G}(\text{population}[1@1])
\end{aligned}$$

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 ‘composition-OfParts’ operator, which translates to Core Forms/3’s explicit use of that operator. Hence, its guarantee and requirement sets are derived solely from the cells in its gridROset.

$$\begin{aligned}
\mathrm{R}(\text{population}) &= \mathrm{R}(\text{population}[1@1]) \cup \mathrm{R}(\text{population}[2@1]) \cup \mathrm{R}(\text{population}[3@1]) \\
&\quad \cup \mathrm{R}(\text{population}[4@1]) && [\text{RM}] \\
&= \text{primitiveNumberOperations} && [\text{Table 7}] \\
\mathrm{G}(\text{population}) &= \mathrm{G}(\text{population}[1@1]) \cap \mathrm{G}(\text{population}[2@1]) \\
&\quad \cap \mathrm{G}(\text{population}[3@1]) \\
&\quad \cap \mathrm{G}(\text{population}[4@1]) && [\text{GM}] \\
&= \text{primitiveNumberOperations} && [\text{GC}]
\end{aligned}$$

$$\begin{aligned}
\text{Thus, } \mathrm{R}(\text{population}) &= \text{primitiveNumberOperations} \\
&\subseteq \text{primitiveNumberOperations} \\
&= \mathrm{G}(\text{population})
\end{aligned}$$

In the same way, the matrix location can be shown to have an empty requirement set and a guarantee set of primitiveTextOperations 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. They all have the same requirement and guarantee sets, so we give only one example here.

$$\begin{aligned}
\mathrm{R}(\text{location}[\text{numrows}]) &= \text{primitiveNumberOperations} && [\text{RN}] \\
\mathrm{G}(\text{location}[\text{numrows}]) &= \mathrm{G}(4) = \text{primitiveNumberOperations} && [\text{GC}]
\end{aligned}$$

$$\begin{aligned}
\text{Thus, } \mathrm{R}(\text{location}[\text{numrows}]) &= \text{primitiveNumberOperations} \\
&\subseteq \text{primitiveNumberOperations} \\
&= \mathrm{G}(\text{location}[\text{numrows}])
\end{aligned}$$

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.

Now suppose a new cell X were added to form population, where X's formula was 'population : location[1@1] + 5'. A type error would occur because the matrix cell location[1@1] guarantees primitiveTextOperations, whereas it is required to support primitiveNumberOperations. More precisely, in Core Forms/3 terms, since R(location[1@1]) = R(+ 5 : plusA) = primitiveNumberOperations, where +5 is the appropriate copy of + for adding 5, and since G(location[1@1]) = primitiveTextOperations, then R(location[1@1]) $\nsubseteq$ G(location[1@1]), and hence the program would not be type safe.

5. Adding Inheritance

5.1. Similarity Inheritance

This section provides necessary background in similarity inheritance for showing in ensuing sections how the model of types can be extended to support similarity inheritance, and thereby to support traditional forms of inheritance as well.

Similarity inheritance [1] is a new, fine-grained model of inheritance intended for responsive, declarative VPLs. From the perspective of type theory, the most important difference from other forms of inheritance is the fact that it allows inheritance of entire types as well as of individual operations. From the perspective of the VPL user, another important difference from traditional forms of inheritance is that similarity inheritance's basic relation ('like') is about implementation similarities only, instead of traditional inheritance's 'is-a' relation, which is about both implementation and abstract similarities. For example, with similarity inheritance, it is sensible to say that a Queue is 'like' a Stack in order to share some of the implementation, but with traditional inheritance, sharing code in this particular way would be considered to be bad design (since a Queue is not a Stack in an abstract sense). Similarity inheritance is intended for users who are not trained in object oriented programming, supporting their reuse of code by allowing them to customize starting from *any* existing example, as in copy/paste, but without the copy/paste disadvantage of losing all the underlying relationships.

Because similarity inheritance is specifically for responsive VPLs, its definition requires both a model of interaction (between the programmer and the computer) and a semantic model. The interaction model is composed of the tuple (**C**, **F**, **L**, **R**), where **C** is the user copy action that creates an inherited definition, **F** is the formula definition action, **L** is a liveness level 3 or higher from Tanimoto's [29] liveness scale indicating that immediate semantic feedback is automatically provided, and **R** is a representation mechanism that explicitly includes all inherited formulas and inheritance relationships in each object's representation. Due to these characteristics of 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 itself rather than also requiring the inspection of parent objects or descriptive classes. The implication of the presence of element **L** is that the programmer creates and manipulates live objects while constructing the program, rather than abstract

Table 8. Conditions that establish and remove an inheritance ($\rightarrow$) relationship. (Postconditions that duplicate preconditions are omitted.) The first row defines large-grained similarity, and means that if **C** is applied to a VADT form F, an inheritance relationship will be created between F and a new VADT form G such that all of F's operations are inherited by G (this can be abbreviated as $F \rightarrow G$). The second row defines fine-grained similarity, which allows a single operation $F:Op_k$ to be copied to VADT form G to create a similarity relationship between $F:Op_k$ and $G:Op_k$. The third row implies that overriding removes 'upstream' inheritance relationships

Action	Precondition	Postcondition
C applied to F	F is a VADT form; G does not exist	G is a new VADT form $\forall F:Op_i \in F.ROset$, $F:Op_i \rightarrow G:Op_i$
C applied to $F:Op_k$ and G	F and G are VADT forms, $F \neq G$	$F:Op_k \rightarrow G:Op_k$
F applied to $G:Op_k$	G is a VADT form	$\forall F,\ F:Op_k \not\rightarrow G:Op_k$

descriptions of objects. The details of elements **L** and **R** have little impact on type theory, so will largely be ignored in this paper.

The semantic model is as follows. Each type definition in a program has a set of operations $\{Op_1, Op_2, \ldots, Op_n\}$. In Forms/3 and Core Forms/3, a type definition is a VADT form F. The definition of each operation $F:Op_i$ is provided by the formula in cell or cell group $F:Op_i$ in F.ROset. The symbol $\rightarrow$ denotes the inheritance relation supported by similarity inheritance. For example, $A \rightarrow C$ denotes C inheriting from A; we say C is 'like' A. (The arrow points from the original version to the inherited version, which maintains our convention in this paper of having arrows point in the direction of information flow). It is also useful to define the transitive closure of inheritance, denoted $\overset{+}{\rightarrow}$: $A \overset{+}{\rightarrow} C$ iff $A \rightarrow C$, or $A \rightarrow B$ and $B \overset{+}{\rightarrow} C$.

The actions **C** and **F** determine when the inheritance relation holds, as summarized in Table 8.

As part of applying action **C** to a form F, the user may rename (provide a new formID for) the resulting copy, and doing so designates the resulting copy as a *promoted copy*. The only formal difference between an ordinary copy F1 and a promoted copy F2 is that F2.modelName $\neq$ F.modelName; rather, F2.modelName = F2.formID. From the user's standpoint, having a user-specified name/formID is useful if the intent of the copy was to start a new type (which is the only application of similarity inheritance we discuss in this paper; see Djang and Burnett [1] for other applications). From the standpoint of type reasoning, promoting copy F2 reduces the type information propagated about F2 to that which can be inferred from its operations, because F2's new modelName prevents use of the [Copies] axiom on F2.

Similarity inheritance supersedes single inheritance and multiple inheritance. Regarding single inheritance, $F \rightarrow G$ is the same relationship as if a class (or object) F is subclassed (or cloned) under another G, in the class-based and prototype-based models. These models support similarity inheritance's concept of 'like' ($\rightarrow$) at the granularity of entire types (classes or objects), and 'not like' ($\not\rightarrow$) at the granularity of operations through overriding. However, neither supports 'like' at the granularity of operations, that is, the ability to inherit just one operation. Multiple and even mutual inheritance are direct by-products of this fine-grained capability of similarity inheritance. Multiple inheritance occurs in cases such as $F \rightarrow G$ and $H : Op_1 \rightarrow G : Op_1$, and mutual inheritance occurs in cases such as $G : Op_2 \rightarrow H : Op_2$ and $H : Op_3 \rightarrow G : Op_3$.

5.2. Forms/3 Examples

In Forms/3, the interaction model is instantiated as follows. Action **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 and cell groups onto the form. Recall that **C** is not simply an edit action, but establishes an inheritance relationship. Action **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 cell and cell group, whether copied or not, is visible, which allows it to be edited by action **F**. Shading indicates whether a form or cell/cell group is copied. (See Djang and Burnett [1] for additional details of the representation.)

Suppose the Forms/3 programmer wants to make use of an existing implementation of type Stack to create a new type that is similar to a stack, for example, a queue. The programmer can start with a copy of the Stack form in Figure 8 and then modify it using action **F**. Inheritance and overriding of cell/cell group names are also allowed and, since inheritance is fine grained, it is also possible to delete cells/cell groups from the copy. In this example, a change to the formula for the push abstraction box and some renaming of cells and cell groups are all that is required to turn the copy into a Queue (see Figure 9). Because Stack $\rightarrow$ Queue, changes to formulas on Stack will propagate to Queue unless they have been overridden. For example, a bug fix to the formula for push on Stack would not have any effect on Queue, but a bug fix to pop's formula would propagate to dequeue on the Queue form.

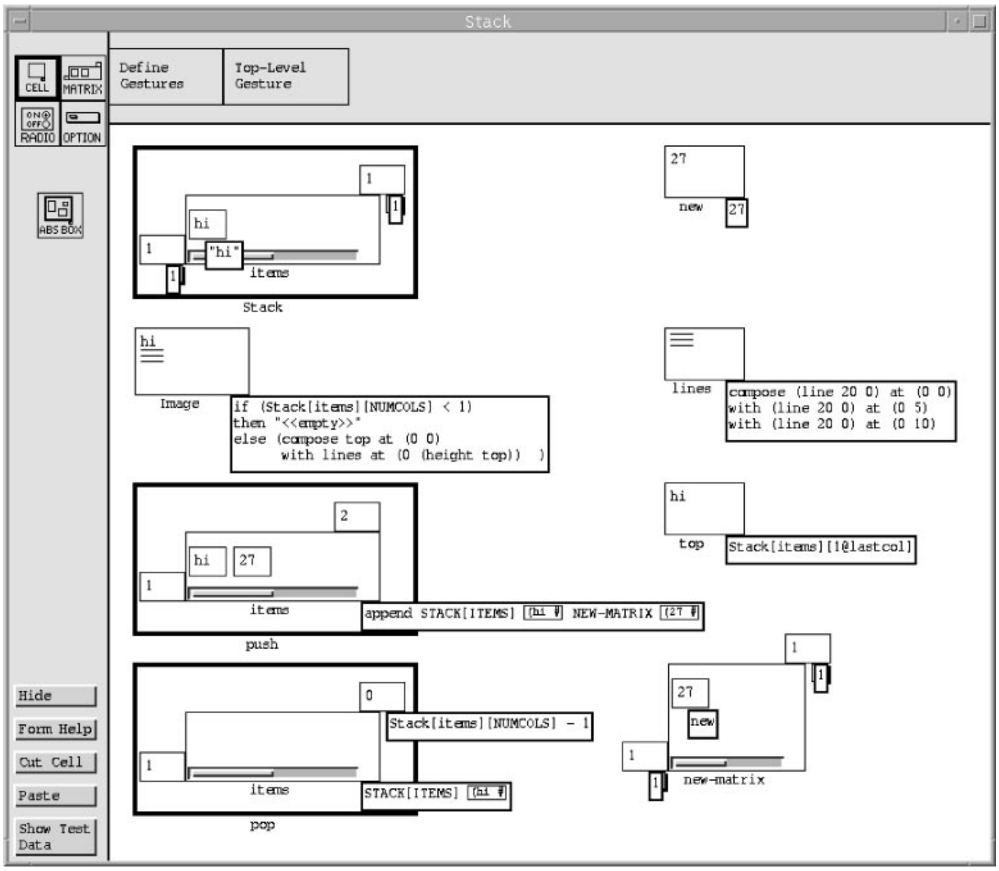
Now suppose, as in Figure 10, someone has added the new operations size and empty? to Queue. Another programmer might find those operations useful for Stack as well and copy them to the Stack form. Stack and Queue now both inherit from each other.

5.3. Type Inference with Similarity Inheritance

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

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

This new version says that the guarantee set includes not only every operation defined on form F, but also the word 'like' prepended to every operation from which the operations on form F are inherited. For example, since in Figure 9, Stack → Queue, then G(Queue : Queue) includes not only the operations on form Queue, but also the word 'like' with the Stack operation top, the word 'like' with the Stack operation pop, and so



```
if (Stack[items][NUMCOLS] < 1)
then "<<empty>>"
else (compose top at (0 0)
      with lines at (0 (height top))  )
```

```
compose (line 20 0) at (0 0)
with (line 20 0) at (0 5)
with (line 20 0) at (0 10)
```

```
Stack[items][1@lastcol]
```

```
append STACK[ITEMS] [hi #] NEW-MATRIX [27 #]
```

```
Stack[items][NUMCOLS] - 1
```

```
STACK[ITEMS] [hi #]
```

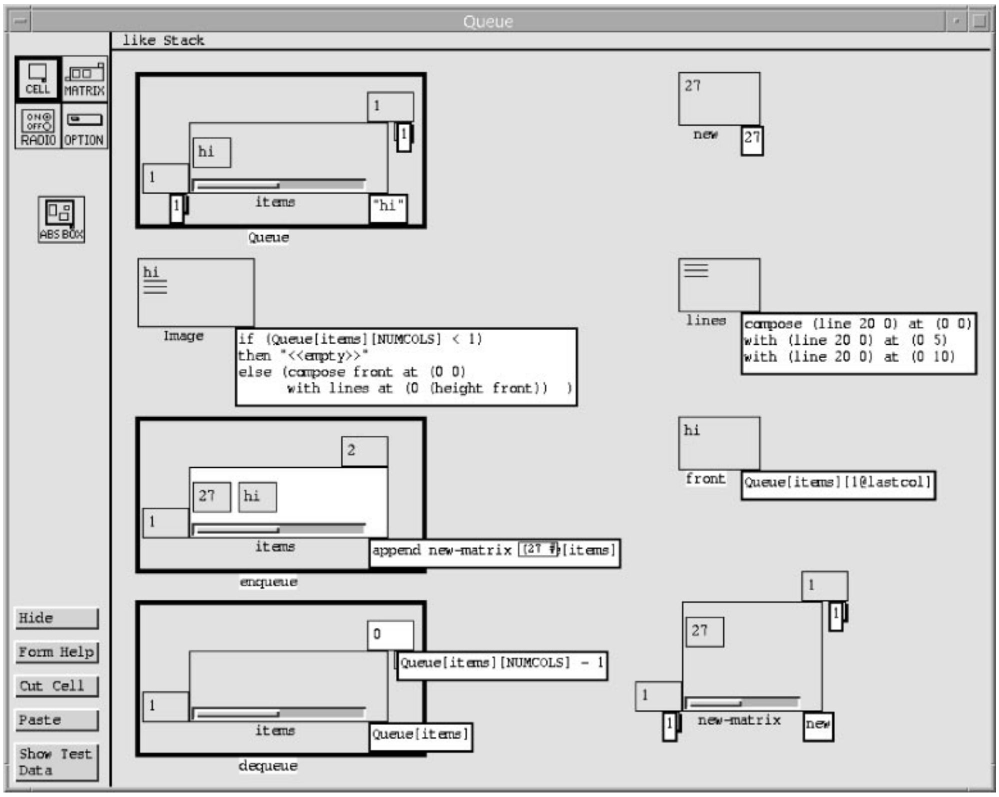
Figure 8. The Stack form

on. Inherited operations are already included on form F (recall the self sufficiency property), but the user is allowed to rename them, such as by changing the name of inherited cell top to front on the Queue form, so by explicitly including the "like' top' entry, the operation is known to the inference system by all of its aliases.

A revised definition of type safety is now needed that takes into account the 'like' entries in guarantee sets:

Revised Definition. If ∀ROs X $\overset{+}{\in}$ program P and ∀Op ∈ R(X) ⇒ either Op ∈ G(X) or 'like' Op ∈ G(X), then P is *type safe*.

In essence, this definition says that every required operation needs to either be present in the guarantee set, or the operation from which it is inherited, prepended with the word 'like', needs to be present in the guarantee set. As a corollary to this revised definition, a type error is now defined as: ∃X, Op such that Op ∈ R(X), Op ∉ G(X), and 'like' Op ∉ G(X).



```
if (Queue[items][NUMCOLS] < 1)
then "<<empty>>"
else (compose front at (0 0)
      with lines at (0 (height front))  )
```

```
compose (line 20 0) at (0 0)
with (line 20 0) at (0 5)
with (line 20 0) at (0 10)
```

```
Queue[items][1@lastcol]
```

```
append new-matrix [27 ?][items]
```

```
Queue[items][NUMCOLS] - 1
```

```
Queue[items]
```

Figure 9. A Queue form created with similarity inheritance from the Stack form of Figure 8, as indicated by "like stacks" displayed upper left. Several names and one matrix are unshaded to indicate that they have been overridden

An obvious implication of axiom [GA′] is that, since the guarantee sets are larger than those inferrable under the pre-inheritance version (axiom [GA]), more programs can potentially be inferred to be type safe than under [GA].

5.4. Operation Polymorphism

In addition to supporting code reuse via inheritance as demonstrated to this point, similarity inheritance also supports operation polymorphism, similar to dynamic dispatch in other languages. That this capability, which sounds inherently dynamic on the surface, can coexist with static type inference is due to the fact that type reasoning is done at the granularity of operations.

Suppose that a cell Y results in an instance of the Stack type in Figure 8, that a cell Z references cell empty? on the appropriate copy of Stack (i.e., Z references Stack_Y : empty?), and that the program is type safe. Now suppose that the user edits cell Y so that its value is an instance of Queue (Figure 9). This would mean that Stack_Y : Stack now references a Queue, which would violate axiom [AbsType] in Table 4.

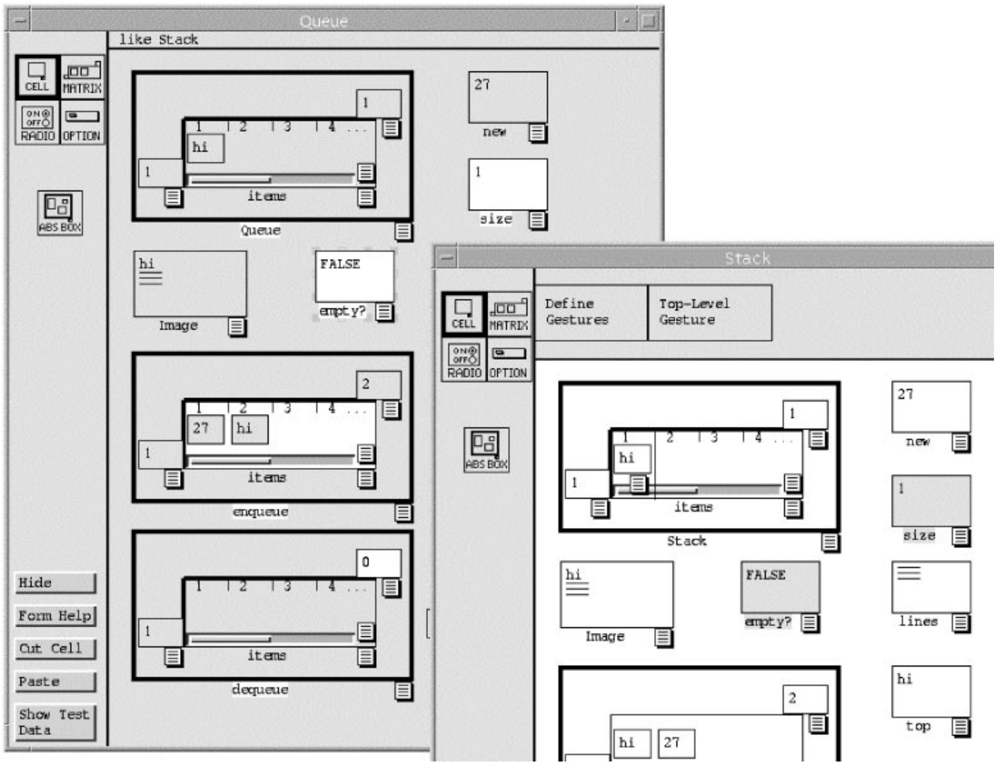


Figure 10. Mutual inheritance between Queue and Stack. The new cells size and empty? appear white on the Queue form where they originated, and shaded on the Stack form

The implications of axiom [AbsType] are that if one abstraction box has a dataflow path to another abstraction box, both must be on forms with the same modelName; because of [VADTformExistsC], the same constraint exists if a constant is the source in the dataflow chain. The violation of this axiom can be detected statically, because it involves a static dataflow chain either from a constant to an abstraction box or from one abstraction box to another. When a formula edit such as the one to cell Y causes a violation of this axiom, the system automatically generalizes the monomorphic reference in cell Z to transform it into a *polymorphic reference*. This generalization, which is an extension of the generalization of Yang and Burnett [42], happens only when a cell is edited. The result is that Z's formula is changed from 'Stack$_Y$: empty?' to '$\langle$VADT$\rangle_{\mathrm{Y}}$: empty?', namely a reference to cell empty? on whatever VADT form is appropriate for Y, which can be determined statically from the dataflow path leading to Y. Generalizing Z's formula in this way removes the violation of the axiom, allowing type inference to proceed in the usual way to make sure the guarantees for Y (which are now the Queue operations, by virtue of Y's new formula) fulfill the requirements of Y ({empty?}, the same set as before the edit to Y, since Z still references a cell of that name).

This generalization not only allows the edit in which Y references a Queue, it also would allow Y to be edited to polymorphically reference the result of a polymorphic ‘if’ or a lazy matrix while still immediately verifying type safety.

5.5. Example: Type Inference in the Presence of Single Inheritance

The Stack and Queue example at the stage depicted by Figures 8 and 9 can show how our model of types works in the presence of single inheritance. First consider what the main abstraction box on each form guarantees. The guarantee sets for these abstraction boxes are lengthy, because they include every RO $\in$ the ROset for the forms. (Note that type safety is separated from information hiding, which is solved in a different way [43].) 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.

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′]

As we briefly mentioned before, since form Queue was created via similarity inheritance from form Stack, the guarantee sets for abstraction boxes on a Queue form include several operations inherited from Stack (‘like’ pop, ‘like’ top, etc.). Notice, however, that these guarantee sets do not include ‘like’ push’, because the programmer overrode the similarity between Stack’s push and Queue’s enqueue.

Figure 11 shows a simple form with some uses of operations on a stack. Cell collection references a Stack, so its guarantee set is the same as G(Stack:Stack). Assuming that cells removed-item and the-rest are the only references to cells that perform operations on cell collection, cell collection’s requirements set contains two operations:

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

Since G(Queue) on every copy of form Queue includes “like’ top’ and “like’ pop’, a reference to Queue on one of these copies by cell collection would have triggered generalization of the-rest and removed-item, thereby preserving type safety according to the revised definition of type safety in Section 5.3.

5.6. Example: Type Inference in the Presence of Multiple Inheritance

The Deque (double-ended queue) at the stage of development shown in Figure 12 illustrates use of multiple inheritance. Deque inherits most of its ROset from Queue, but

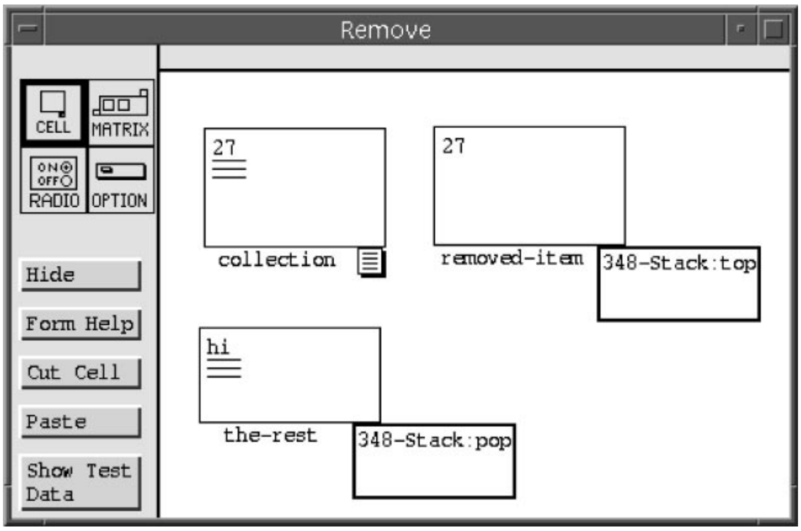


Figure 11. Cells removed-item and the-rest contain examples of references that will eventually become polymorphic. The figure shows the formulas before generalization. Cell collection's formula (not shown) references some instance of a Stack, and form 348-Stack's main abstraction box, Stack, in turn references cell collection. If cell collection's formula is changed to reference, for example, a Queue, the formulas of removed-item and the-rest will be generalized

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.

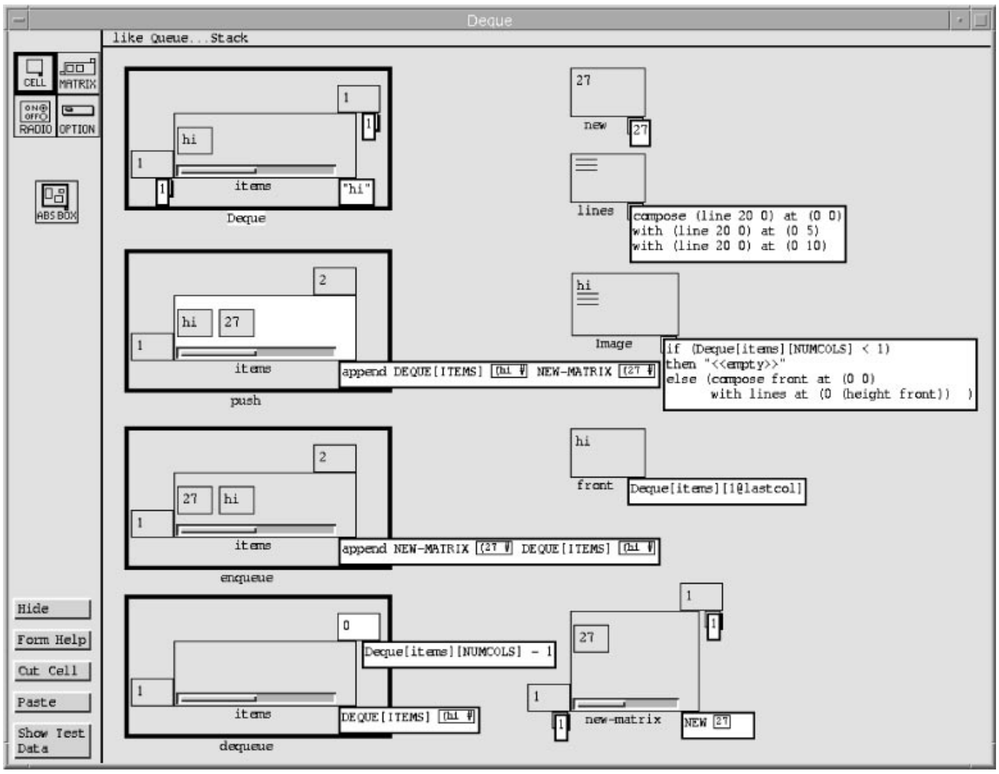
G(Deque:Deque) = {Deque, enqueue, dequeue, front, Image, new, lines, new-matrix, Deque[items], Deque[items][numrows], Deque[items[1@1], …, 'like' Stack, 'like' pop, 'like' top, 'like' Stack[items], 'like' Stack[items][numrows], …, push, 'like' Queue, 'like' Queue[items], 'like' Queue[items][numrows],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

6. Properties of the Types Model

6.1. Soundness

Most other type systems reason about the types themselves, and in these systems the notion of soundness reduces to whether the inferred type *name* is always correct. However, our type system is based on operation guarantees and operation requirements as a way of reasoning directly about type *safety*. Hence, a notion of soundness with respect to type safety is needed.



```
compose (line 20 0) at (0 0)
with (line 20 0) at (0 5)
with (line 20 0) at (0 10)
```

```
if (Deque[items][NUMCOLS] < 1)
then "<<empty>>"
else (compose front at (0 0)
      with lines at (0 (height front)) )
```

```
append DEQUE[ITEMS] (hi NEW-MATRIX (27
```

```
Deque[items][1@lastcol]
```

```
append NEW-MATRIX (27 DEQUE[ITEMS] (hi
```

```
Deque[items][NUMCOLS] - 1
```

```
DEQUE[ITEMS] (hi
```

```
NEW 27
```

Figure 12. A Deque form inheriting from the forms and operations for Stack and Queue

Let **A** be the axiom set presented in this paper. We define soundness with respect to type safety as follows:

Definition. A is *sound with respect to type safety* iff applications of **A** cannot prove that a program P is type safe if P's dynamic execution could result in type errors.

Type safety has earlier been defined in terms of guarantee sets and requirement sets. Substituting this for the phrase 'a program P is type safe' in the above definition produces the following:

(1) **A** is sound with respect to type safety iff applications of **A** cannot prove that

$\forall$ROs X $\overset{+}{\in}$ program P and $\forall \mathrm{Op} \in R(X)$, either $\mathrm{Op} \in G(X)$ or 'like' $\mathrm{Op} \in G(X)$ if P's dynamic execution could result in type errors.

Focusing now on the last line of (1), P's dynamic execution could result in type errors if (and only if), despite the contents of G(X) and R(X), in some dynamic execution of P an operation could be used on some X that was not actually guaranteed for X. Let $G_{\mathrm{ideal}}(X)$ and $R_{\mathrm{ideal}}(X)$ be 'ideal' sets that perfectly reflect the operations on X that are actually guaranteed and required, respectively, in any dynamic execution of P. Wording the last line of (1) in terms of these ideal sets gives:

(2) **A** is sound with respect to type safety iff applications of **A** cannot prove that $\forall$ROs X $\overset{+}{\in}$ program P and $\forall \mathrm{Op} \in \mathrm{R(X)}$, either $\mathrm{Op} \in \mathrm{G(X)}$ or 'like' $\mathrm{Op} \in \mathrm{G(X)}$ if $\exists \mathrm{X} \overset{+}{\in}$ program P and $\exists \mathrm{Op} \in \mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$, such that $\mathrm{Op} \notin \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$ and 'like' $\mathrm{Op} \notin \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$.

From (2), it is clear that if $\mathrm{G(X)} = \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$ and $\mathrm{R(X)} = \mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$, then it follows that **A** is sound with respect to type safety. However, it is more convenient for proof purposes to subdivide the property of equality to an ideal set into two properties, termed reliability and comprehensiveness.

We define G(X) to be *reliable* iff $\mathrm{G(X)} \subseteq \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$, and similarly define R(X) to be reliable iff $\mathrm{R(X)} \subseteq \mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$. We define G(X) to be *comprehensive* iff $\mathrm{G(X)} \supseteq \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$, and similarly define R(X) to be comprehensive iff $\mathrm{R(X)} \supseteq \mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$. Of course, if G(X) is both reliable and comprehensive then $\mathrm{G(X)} = \mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$, and likewise for R(X). Intuitively, G(X), for example, is reliable if it contains *only* operations that are actually supported for X, and is comprehensive if it includes *every* operation that is actually supported for X.

Thus, to this point, we have shown that proving the guarantee sets to be both reliable and comprehensive and the requirement sets to be both reliable and comprehensive would be sufficient to prove that **A** is sound with respect to type safety. However, it turns out that it is not necessary to meet all four of these conditions. In the remainder of this section we complete the proof that **A** is sound with respect to type safety by proving that

1. the guarantee sets are reliable;
2. the requirement sets are comprehensive; and
3. reliability of the guarantee sets and comprehensiveness of the requirement sets are sufficient for soundness with respect to type safety.

6.1.1. Reliability of the Guarantee Sets

We have defined a guarantee set G(X) to be reliable if every operation in G(X) is in $\mathrm{G}_{\mathrm{ideal}}(\mathrm{X})$ by virtue of actually being supported for X. We restate this definition, separately for non-matrices and matrices, in terms of Core Forms/3 programming objects.

(a) For any non-matrix RO X, G(X) is reliable iff every operation in G(X) according to **A** is defined on X's VADT form F, either by being present under the same name, or by having been inherited and renamed.

Proof that G(X) is reliable for any non-matrix RO X:

Case 1: X *is an abstraction box on its VADT form F.*

Let $\mathrm{Op} \in \mathrm{G(X)}$

$\Rightarrow \mathrm{Op} \overset{+}{\in} \mathrm{F.ROset}$ by [GA′]

Hence, since Op is defined on F, $G(X)$ is reliable.

Let 'like' $\mathrm{Op} \in G(X)$

$\Rightarrow \mathrm{Op} \overset{+}{\rightarrow} \mathrm{Op2}$ where $\mathrm{Op2} \overset{+}{\in}$ F.ROset by [GA′]

Hence, since Op is a different name for Op2, which is on F, $G(X)$ is reliable.

Case 2: *X is not an abstraction box, and* $X \leftarrow Z$.

$X \overset{+}{\leftarrow} F:Y$, where $F:Y$ is an abstraction box by [Inst]

X's VADT form F_X is a copy of F by [VADTformExistsR] and [Copies]

$\Rightarrow \forall \mathrm{Op} \overset{+}{\in}$ F.ROset, $\mathrm{Op} \overset{+}{\in} F_X$.ROset by [Copies]

$G(X) = G(F:Y)$ by (repeated application of) [Gref]

$G(F:Y)$ is reliable by Case (1) above

Hence since $G(X) = G(F:Y)$, which is reliable given the same ROset for Y as exists for X, then $G(X)$ is reliable.

Case 3: *X is not an abstraction box, and* $X \leftarrow_c C$. Because constants are shortcuts for referencing an abstraction box on a built-in VADT form, this case trivially reduces to case 2.

(b) For any matrix X, $G(X)$ is reliable iff every operation in $G(X)$ according to **A** is present in $G(X[i])$ and $G(X[i])$ is reliable, $\forall X[i] \in$ X.gridROset.

Proof that $G(X)$ is reliable for any matrix X:

Let $x \in G(X)$ where $x = \mathrm{Op}$ or $x =$ 'like' Op

$\Rightarrow x \in G(X[i])$, $\forall X[i] \in$ X.gridROset by [GM]

$X[i]$ is not a matrix by definition of matrix ROsets (Section 3.2)

$\Rightarrow G(X[i])$ is reliable by (a) above

Hence, $G(X)$ is reliable.

6.1.2. Comprehensiveness of the Requirement Sets

We have defined a requirement set $R(X)$ to be comprehensive if $R(X)$ includes every operation that will be used on X in some dynamic execution of P. The restatement in terms of Core Forms/3 programming objects again is separated for non-matrices and matrices.

(a) For any non-matrix RO X, $R(X)$ is comprehensive iff, given $\mathrm{Op} \notin R(X)$ according to **A**, then there is no *use* of Op on X (that is, $\nexists\, Z$ such that $Z \leftarrow F_X:\mathrm{Op}$), and there is no use of Op on any Y (that is, $\nexists\, Z$ such that $Z \leftarrow F_Y:\mathrm{Op}$) such that $Y \overset{+}{\leftarrow} X$.

Proof that R(X) is comprehensive for any non-matrix RO X (by contradiction): Assume that $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$. Then either case 1 or case 2 below must be true:

Case 1: $\exists Z$ *such that* $Z \leftarrow F_X{:}Op$

$\Rightarrow \mathrm{Op} \in \mathrm{R}(\mathrm{X})$ by [R1]

But $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$ by the assumption above.

Case 2: $\exists Z$ *such that* $Z \leftarrow F_Y{:}\ Op$ *and* $Y \overset{+}{\leftarrow} X$

$\Rightarrow \mathrm{Op} \in \mathrm{R}(\mathrm{Y}) \Rightarrow \mathrm{Op} \in \mathrm{R}(\mathrm{X})$ by [R1]

But $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$ by the assumption above.

Hence, R(X) is comprehensive.

(b) For any matrix X, R(X) is comprehensive iff, given $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$ according to A, then there is no use of Op on any X[i] ($\nexists$Z such that $\mathrm{Z} \leftarrow \mathrm{F}_{\mathrm{X[i]}} : \mathrm{Op}$).

Proof that R(X) is comprehensive for any matrix X (by contradiction): Assume that $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$, and that $\exists$Z such that $\mathrm{Z} \leftarrow \mathrm{F}_{\mathrm{X[i]}} : \mathrm{Op}$.

$\Rightarrow \mathrm{Op} \in \mathrm{R}(\mathrm{X[i]})$ by [R1]

$\Rightarrow \mathrm{Op} \in \mathrm{R}(\mathrm{X})$ by [RM]

But $\mathrm{Op} \notin \mathrm{R}(\mathrm{X})$ by the assumption above.

Hence, R(X) is comprehensive.

6.1.3. Sufficiency

We have pointed out that all four properties (reliability of guarantee sets, comprehensiveness of guarantee sets, reliability of requirement sets, and comprehensiveness of requirement sets) would have been sufficient to show that **A** is sound with respect to type safety. We now show that the two properties just proven, reliability of guarantee sets and comprehensiveness of requirement sets, are sufficient for soundness with respect to type safety, i.e. that the other two properties are not necessary.

Focusing first on the requirement sets, let R(X) be the requirements on X according to **A**, which we have just shown to be comprehensive. R(X) is not reliable if it is possible for Op to be in R(X) when Op can never be executed for Y. For example, suppose P (in full Forms/3) consists of cells X and Y, and that Y.formula = 'If true = false then (X div 2) else 3'; from this, R(X) = primitiveNumberOperations, in order to ensure divisibility. However, since the division can never actually execute then $\mathrm{R}_{\mathrm{ideal}}(\mathrm{X}) = \{\}$. Comprehensiveness of requirements R(X) means that $\mathrm{R}(\mathrm{X}) \supseteq \mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$. Hence, whereas comprehensiveness without reliability can allow too many requirements to be inferred, it cannot allow requirements in $\mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$ to be omitted from R(X). Thus, given comprehensive requirements sets, every program inferred to be type safe using R(X) would also be type safe using $\mathrm{R}_{\mathrm{ideal}}(\mathrm{X})$, and reliability of the requirement sets is not necessary to achieve soundness with respect to type safety.

By similar reasoning, it is not necessary for guarantee sets to be comprehensive given that they are reliable: if G(X) is not comprehensive, its missing guarantees will result in an overly conservative measure of which programs are type safe, but will not allow programs deemed unsafe using $G_{ideal}(X)$ to be deemed type safe using $G(X)$.

6.2. Understandability of User Communication

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 professional programmers. Although the question of understandability by a particular audience ultimately requires testing human subjects' 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 communicate with the user to be the type error messages, since without error messages, 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 (e.g. primitiveCircleOperations), formIDs (e.g. Stack) and cellIDs (e.g. empty?). In fact, since the concept of primitiveOperations is a shorthand for the specific set of operations, it could be eliminated from user communications, reducing the number of concepts in the vocabulary of types to only two. Since user-assigned names are used in place of generated IDs, we expect these concepts to be familiar and thus understandable to the user. Because of this simple type vocabulary, we believe type inference under our model of types offers a potential for greater understandability than type inference with other models of types. In particular, the absence of polymorphic types, function types, and compositions of types, simplifies the type vocabulary required of the user, whereas type models that use these concepts must also communicate about them when type errors occur, complicating the type vocabulary with which the user must be conversant.

7. Future Work

Although similarity inheritance has been implemented, the static type system to support it presented in this paper is so far solely a theoretical model, not an implementation. Our next step will be to devise algorithms that will allow this model to be instantiated in the Forms/3 implementation. (Forms/3 itself is currently implemented in Lisp under the Garnet user interface system, and is freely available at http://www.cs.orst.edu/~burnett.) An important goal of the implementation of the type system will be to provide insights into the efficiency characteristics of the model. Also, after the type system has been implemented, we hope to conduct user experiments to see if automatic type inference under this fine-grained model of types is indeed useful and helpful to various types of VPL users.

8. Conclusion

Static type inference increases the amount of immediate visual feedback that VPLs can potentially provide, because it affords the possibility of immediate (edit-time) feedback as soon as the user introduces a type error. In this paper, we have presented a static type inference system intended for the purpose of type checking in first-order declarative VPLs, both those for programmers and those for end users. The model is presented here in the context of one particular such VPL, Forms/3. 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. First, they allow support even of flexible, fine-grained approaches to inheritance, which supersedes support of traditional, coarse-grained approaches to inheritance. Second, they allow support for these various forms of inheritance without the traditional measure of re-introducing explicit type declarations. Finally, they allow 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 simplifies the user vocabulary associated with use of this model. These properties, while potentially useful in traditional programming languages, are critical in declarative VPLs, because their absence has prevented the possibility of using static type inference either in VPLs incorporating similarity inheritance or in VPLs aimed at end users.

Acknowledgments

We would like to thank the members of our research group for their work on the Forms/3 design and implementation and for their feedback on the model of types. We are also grateful to the reviewers of an earlier draft of this paper for their extremely helpful comments and suggestions. This work has been supported by: Hewlett-Packard; Harlequin; a NASA Graduate Student Researcher Award and the National Science Foundation under grants CCR-9308649, ASC-9523629, and an NSF Young Investigator Award.

References

1. R. W. Djang & M. Burnett (1998) Similarity inheritance: a new model of inheritance for spreadsheet VPLs. *1998 IEEE Symposium on Visual Languages*. Halifax, Nova Scotia, Canada, 1–4 September.
2. L. Cardelli & P. Wegner (1985) On understanding types, data abstraction, and polymorphism. *Computing Surveys* **17,** 471–522.
3. M. I. Schwartzbach (1997) Object-oriented type systems: principles and applications. Lecture Notes Efteruddanelseskursus for Datamatikerlaererforeningen, May 14–16. Available at http://www.daimi.aau.dk/~ mis/ootspa.ps.

4. L. Cardelli (1987) Basic polymorphic typechecking. *Science of Computer Programming* **8,** 147–172.
5. Y. Jun & G. Michaelson (1999) A visualization of polymorphic type checking. *Journal of Functional Programming* (to appear).
6. G. Blair, J. Gallagher & J. Malik (1989) Genericity vs inheritance vs delegation vs conformance vs .... *Journal of Object-Oriented Programming* 11–17.
7. A. Goldberg & D. Robson (1983) *Smalltalk-80 The Language and its Implementation.* Addison-Wesley, Reading, MA.
8. D. Ungar & R. B. Smith (1987) Self: the power of simplicity. *Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*). 4–8 October, pp. 227–242.
9. J. Palsberg & M. Schwartzbach (1991) Object-oriented type inference. *OOPSLA'91.* Phoenix, AZ, 6–11 October, pp. 146–161.
10. O. Agesen, J. Palsberg & M. Schwartzbach (1993) Type inference in SELF: analysis of objects with dynamic and multiple inheritance. *ECOOP'93 European Conference on Object Oriented Programming.* Lecture Notes in Computer Science, Vol. 707. Springer, Berlin, pp. 247–267.
11. W. Cook, W. Hill & P. Canning (1990) Inheritance is not subtyping. *POPL'90.* San Francisco, CA, pp. 125–135.
12. W. Harris (1991) Contravariance for the rest of us. *Journal of Object-Oriented Programming* **4,** 10–18.
13. B. H. Liskov & J. M. Wing (1994) A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems* **16,** 1811–1841.
14. R. W. Sebesta (1996) *Concepts of Programming Languages*, 3rd edn. Addison-Wesley, Menlo Park, CA.
15. B. Meyer (1992) *Eiffel: The Language.* Prentice-Hall, Englewood Cliffs: NJ.
16. B. Stroustrup (1992) *The C++ Programming Language*, 2nd edn. Addison-Wesley, Reading, MA.
17. P. Canning, W. Cook, W. Hill, W. Olthoff & J. C. Mitchell (1989) F-bounded polymorphism for object oriented programming. *Proceedings International Conference on Functional Programming Languages and Computer Architecture.* September, pp. 273–280.
18. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group: G. T. Leavens & B. Pierce (1995) On binary methods. *Theory and Practice of Object Systems*, pp. 221–242.
19. K. Bruce, L. Petersen & A. Fiech (1997) Subtyping is not a good 'Match' for object-oriented languages. In *ECOOP Proceedings. Lecture Notes in Computer Sciencs*, Vol. 1241. Springer, Berlin, 104–127.
20. S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman & P. Wadler (1999) *Haskell 98: A Non-strict, Purely Functional Language.* http://haskell.cs.yale.edu/onlinereport/, February 1. (Updated version of: Paul Hudak *et al.* (1992) Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2, *ACM SIGPLAN Notices*, **27**, May.)
21. R. Milner (1978) A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17,** 348–375.
22. M. Wand (1986) Finding the source of type errors. *13th Annual ACM Symposium on Principles of Programming Languages.* St. Petersburg Beach, FL, January 13–15, pp. 38–43.
23. F. Bent & D. Duggan (1996) Explaining type inference. *Science of Computer Programming* **27,** 37–83.
24. P. T. Cox, F. R. Giles & T. Pietrzykowski (1989) Prograph: a step towards liberating programming from textual conditioning. *IEEE Workshop on Visual Languages.* Rome, Italy, October, pp. 150–156.
25. A. Cypher & D. Smith (1995) KidSim: end user programming of simulations. *Proceedings of CHI'95: Human Factors in Computing Systems.* Denver, Colorado, 7–11 May, pp. 27–34.
26. D. Kurlander (1993) Chimera: example-based graphical editing. In: *Watch What I Do: Programming by Demonstration* (A. Cypher, ed.). MIT Press, Cambridge, MA.
27. W. Citrin, M. Doherty & B. Zorn (1997) A graphical semantics for graphical transformation languages. *Journal of Visual Languages and Computing* **8,** 147–173.

28. A. Ambler & A. Broman (1998) Formulate solution to the visual programming challenge. *Journal of Visual Languages and Computing* **9,** 171–209.
29. S. Tanimoto (1990) VIVA: a visual language for image processing. *Journal of Visual Languages and Computing* **2,** 127–139.
30. M. Najork & E. Golin (1990) Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. *IEEE Workshop on Visual Languages*. Skokie, Illinois, October.
31. M. Najork (1996) Programming in three dimensions. *Journal of Visual Languages and Computing* **7,** 219–242.
32. T. D. Kimura, J. W. Choi & J. M. Mack (1990) Show and Tell. *Visual Computing Environments* (E. P. Glinert ed.). IEEE Computer Society Press, Washington, DC.
33. J. Poswig, K. Teves, G. Vrankar & C. Moraga (1992) VisaVis-contributions to practice and theory of highly interactive visual languages. *IEEE Workshop on Visual Languages*. Seattle, Washington, September, pp. 155–162.
34. J. Poswig & C. Moraga (1993) Incremental type systems and implicit parametric overloading in visual languages. *IEEE Symposium on Visual Languages*, August 24–27, pp. 126–133.
35. L. Braine & C. Clack (1996) Introducing CLOVER: an object-oriented functional language. *Implementation of Functional Languages 8th International Workshop*, *IFL'96* (W. Kluge, ed.). Lecture Notes in Computer Science, Vol. 1268.
36. D. Ingalls, S. Wallace, Y. Chow, F. Ludolph & K. Doyle (1988) Fabrik, a visual programming environment. *OOPSLA*, San Diego, CA, September, pp. 176–190.
37. J.-Y. Vion-Dury & F. Pacull (1997) A structured interactive workspace for a visual configuration language. *1997 IEEE Symposium on Visual Languages*. Capri, Italy, September 23–26, 130–137.
38. M. Burnett (1993) Types and type inference in a visual programming language. *IEEE Symposium on Visual Languages*. Bergen, Norway, 24–27 August, 238–243.
39. M. Burnett & H. Gottfried (1998) Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer–Human Interaction*, March 1–33.
40. M. Burnett, J. Atwood & Z. Welch (1998) Implementing level 4 liveness in declarative visual programming languages. *IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, Canada, 1–4 September, pp. 126–133.
41. J. Atwood, M. Burnett, R. Walpole, E. Wilcox & S. Yang (1996). Steering programs via time travel. *IEEE Symposium on Visual Languages*. Boulder, Colorado, September 3–6, pp. 4–11.
42. S. Yang & M. Burnett (1994) From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships. *1994 IEEE Symposium on Visual Languages*. St. Louis, Missouri, October 4–7, pp. 6–14.
43. M. Burnett & A. Ambler (1994) Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing* **5,** 29–60.