Type Handling in a Fully Integrated Programming and Specification Language

Gregory Kulczycki
Clemson University
Clemson, SC
gregwk@cs.clemson.edu

ABSTRACT

Integrated languages combine formal specification and programming features, and make it possible to specify, implement, and verify programs within the same framework. This paper examines the consequences of this fundamental integration on the type system of a software engineering language, using RESOLVE as an example. It explains why name matching for program types coexists naturally with structural matching for math types. It describes a formulation of set theory and its relationship to the type system. And it poses a variety of discussion questions concerning the use of types and subtypes in the specification portion of the language.

Keywords

Type checking, verification, subtypes, set theory, software engineering

1. INTRODUCTION

Verification of component-based software requires languages that integrate programming and specification features, and types are at the heart of this integration. Programming languages are not suited for specification, and specification languages are not used for implementation. The elements of both languages must be integrated to verify that an implementation is correct with respect to a specification. This requires that programming objects—in particular, types—be described in mathematical terms. A wealth of papers have been written about types and type systems, but these papers invariably focus on types in programming (implementation) languages or types in specification languages. The contribution of this paper is its description of a type system for languages concerned with both implementations and specifications.

The desire to build predictable, component-based software has compelled many in the software verification community to develop integrated languages—languages that combine formal specification with programming. Examples of such unions include JML, Eiffel, RESOLVE/C++[1], Z variants, and Larch variants [3, 7, 9, 15]. Most of these integrated languages have resulted from appending a specification language onto a preexisting programming language. In contrast, RESOLVE [12, 14] has been developed from the beginning as both a specification and a programming language. The language is only one part of the RESOLVE system for predictable software development. The system also includes a framework and discipline for building software that is—among other things—reusable, verifiable, efficient, and understandable. The language is intimately tied to the framework and discipline.

For the past few years the author of this paper has been involved in designing and implementing tools that would bring RESOLVE into the world of practical programming. The current focus of this effort is the development of a RESOLVE compiler. The project is complex, not only because the compiler must deal with a programming and specification language combined, but because ongoing research makes the language a moving target (e.g., performance specification and verification [13]). During the course of writing the compiler we have been forced to refine our ideas about how types should be handled in both mathematical and programming contexts.

This paper addresses the following question: What are the implications for the type system in a language that integrates programming and specification? Using RESOLVE as an example, we look for answers to this question. Section 2 presents the type model of RESOLVE and demonstrates how types are treated in programming and mathematical contexts. Section 3 summarizes Ogden's formulation of set theory in RESOLVE [10] and explains how it relates to types. Finally, section 4 examines a few specific issues involving math types and subtypes.

2. OVERVIEW OF TYPES

An integrated language is much more complex than either a programming language or specification language alone, so simplicity is a primary concern. It is essential to have type matching rules that are easily understandable. A programmer (or compiler) should not have to sift through a myriad of rules and exceptions simply to evaluate the type of an expression.

[1]RESOLVE/C++ uses only the specification portion of the RESOLVE language

```
Concept Stack_Template(type Entry;
        evaluates Max_Depth: Integer);
    uses Std_Integer_Fac, String_Theory;
    requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization ensures S = Λ;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures S = ⟨#E⟩ ∘ #S;

    ...

end Stack_Template;
```

Figure 1: A Concept for Stack

Mathematical and programming elements in the RESOLVE language are kept as distinct as possible. Thus, assertions in requires and ensures clauses[2] of operations are strictly mathematical expressions, and conditions in while loops and if statements are strictly programming expressions. Likewise, all variables and types found in a mathematical expression are math variables and math types, and those found in programming expressions are program variables and program types. This means that the same name has a different type depending on whether it appears in a programming or mathematical context. Furthermore, mathematical expressions and programming expressions are type-checked differently—in mathematical expressions, types are matched according to structure, whereas in programming expressions, they are matched strictly by name.

2.1 Math vs Program Context

Figure 1 shows a RESOLVE specification of a Stack component. This simple example turns out to be sufficiently powerful to illustrate the ideas in this paper. The **Type Family** declaration introduces the program type Stack and gives its mathematical model. We use *Type Family* instead of just *Type* because the concept (and therefore the type) is generic until it is instantiated, so the declaration of Stack here encompasses an entire family of types. In the type family declaration, the left side contains the program type Stack, and the right side contains the math type Str(Entry). The fact that mathematical and programming elements come together in a type declaration underscores the fundamental role that types play in an integrated language. The **exemplar** introduces a variable of type Stack to describe properties that hold for any arbitrary variable of type Stack. For example, the **constraints** clause indicates that the length of any Stack must always be less than Max_Depth.

In the specification of **Operation** Push, parameters E and S are program variables. When a call is made to this operation, the compiler checks that the first argument to Push is of type Entry, and the second argument is of type Stack. When S appears in the requires clause, however, the compiler analyzes it as a math variable. The variable S has been declared as program type Stack, but since the variable occurs in a mathematical context, the compiler instead uses the mathematical model of Stack given in the type family declaration. So the variable S appearing in the requires clause has math type Str(Entry). The rest of the concept is analyzed similarly.

[2]preconditions and postconditions

```
Realization Array_Realiz for Stack_Template;

    Type Stack is represented by Record
            Contents: Array 1..Max_Depth of Entry;
            Top: Integer;
        end;
        conventions 0 ≤ S.Top ≤ Max_Depth;
        correspondence
            Conc.S = $\left(\prod_{i=1}^{|S.Top|} \langle S.Contents(i)\rangle\right)^{Rev}$ ;
        initialization
            S.Top := 0;
        end;

    Procedure Push(alters E: Entry; updates S: Stack);
        ...
    end Push;

    ...

end Array_Realiz;
```

Figure 2: A Realization for Stack

In RESOLVE, like in other model-based languages such as VDM and Z, a handful of math types are used for modeling many different program types. This mirrors scientific disciplines like Physics, where the same mathematical model is used to capture widely different concepts. Different program types such as Stack and Queue may both be modeled using mathematical strings. This makes it convenient to write specifications such as the one shown here:

```
    Operation Stk_Q_Transfer(clears S: Stack;
                                    replaces Q: Queue);
        ensures Q = #S^Rev;
```

2.2 Structural vs Name Matching

The implementation or realization of Stack_Template in Figure 2 introduces a Stack type with a specific programming structure. It indicates how a Stack is represented for this particular realization. The **conventions** clause provides the representation invariant—it indicates which representation states are permitted. The **correspondence** clause, or abstraction relation, shows how this representation is related to the mathematical model of Stack given in the concept. Notice that the correspondence clause contains two variables, S and **Conc**.S, that are not declared directly in this scope. These variables are derived from the special exemplar variable in the concept's type declaration. Figure 3 illustrates what the compiler does when analyzing the declaration of Stack in a realization. It locates the exemplar from the type

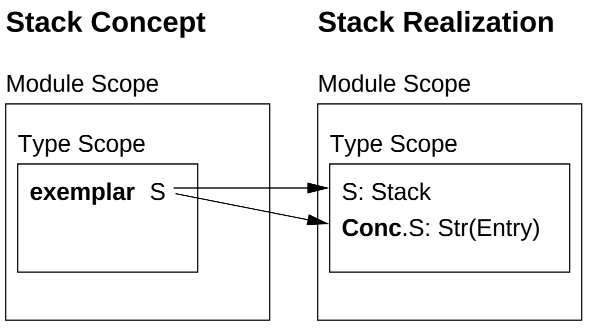

Figure 3: Affect of Exemplar on Realization

family declaration in the corresponding concept, and uses its name to create two variables within the type scope of the realization. The first variable is named S and has program type Stack. The second variable is named **Conc**.S (read as "the conceptual value of S") and has math type Str(Entry), the mathematical model of Stack. The correspondence clause describes the relationship between these two variables.

Program type matching in RESOLVE is done strictly by name. This is reasonable because a primary motivation for introducing different type names is to keep objects of different types distinct. Also, in a language that separates interfaces (as specifications) from implementations, clients will not have access to the structural programming representation of a type, so that structural matching can not be accomplished consistently.

Math type matching is done by structure. The structure consists of math types that can be simple or composite. If a program type name is encountered in a mathematical context, the compiler uses its corresponding mathematical model to convert it to a math type expression. In RESOLVE, like in Z, built-in composite types include set theory operators $\times$, $\rightarrow$, and $\mathcal{P}$. Composite types are parameterized types that take other types as arguments. RESOLVE also permits the use of user-defined composite types. In the type expression Str(Entry), Str is a user-defined composite type (defined in the module String_Theory, which is imported through the **uses** clause in Figure 1). For a composite math type to match another by structure, the types of their arguments must also match. For example, Queue $\times$ Fahrenheit matches Stack $\times$ Centigrade if and only if the mathematical models of Queues and Stacks match, *and* the mathematical models of Fahrenheit and Centigrade match. Constraints on mathematical models—given by the constraint clause in the type family declaration—are ignored by the analyzer; checking constraints is the responsibility of the verifier.

To illustrate the difference between name matching in the programming world and structural matching in the math world, consider the type declaration of Stack in figure 2. The representation uses both a record and an array, which are composite program types. In RESOLVE, the type Record is modeled by a Cartesian product (denoted by the infix operator $\times$), and the type Array is modeled by a function

Table 1: Type Evaluations of Variables

Variable	Program Type	Math Type
S	Stack	$(\mathbb{Z} \rightarrow \text{Entry}) \times \mathbb{Z}$
S.Contents	%Array(10,20)	$\mathbb{Z} \rightarrow$ Entry
S.Contents(1)	Entry	Entry
S.Top	Integer	$\mathbb{Z}$

(denoted by the infix operator $\rightarrow$)[3] [10]. Also, the program type Integer is modeled by the mathematical integers $\mathbb{Z}$.[4] The generic type Entry is treated as a primitive type when seen from a math context because its math model is not known before instantiation.

Now consider a variable S of type Stack. Table 1 shows how a compiler will evaluate the variables in the first column depending on whether they occur in a program or math context. For example, if S.Contents occurs in a requires clause, it evaluates to the math type $\mathbb{Z} \rightarrow$ Entry. If the variable S.Top occurs in the condition of a while loop, it evaluates to the program type Integer. The type %Array(10,20) is a unique name created by the compiler.

A compiler for RESOLVE must keep track of more type information than typical compilers. It must have access to the program name of the type, the program structure of the type, and the math structure of the type. The program structure of the type is not needed for matching purposes, but it is needed to determine whether variables of that type may use the special syntax of Records or Arrays. For example, since the type Stack in the realization above is structurally a record, any variable S of type Stack can use special syntax to refer its fields—S.Contents and S.Top.

3. SET THEORY

Sets are the fundamental building blocks of the RESOLVE language. There are several reasons why sets are a natural choice. First and most importantly, sets are foundational to Mathematics. All programming objects must have a mathematical model, and sets can be used to describe any mathematical domain. No matter how complicated a real world problem is, it can be captured with sets. The same could not be said if we were to use, say, real numbers, as the building blocks of the language. Another reason for using sets is that the basic notions of sets—membership, union, subset, and so forth—are familiar to most students and programmers. Finally, sets are flexible enough to describe the language itself.

3.1 Echelons

The particular flavor of set theory used in RESOLVE has been developed by Bill Ogden at The Ohio State University [10]. The core of the theory is traditional: It starts with the notion of a universe of all sets ($\mathcal{S}et$) and uses the notion of membership ($\in$) as a basis for defining all the operators we expect to see on sets ($\cup$, $\cap$, $\subseteq$, $\mathcal{P}$, $\rightarrow$, etc.). A distinguishing

[3]Strictly speaking, the RESOLVE type Array is modeled by a Cartesian product composed of a function and two integers—one for each bound.

[4]This model will obviously have constraints, involving minimum and maximum values, but recall that constraints are ignored during type-checking

aspect of the theory is the notion of special sets known as *echelons*. Echelons are large universes of sets that are closed under the operations of ordinary set theory, such as unions and power sets.

The motivation for echelons comes from the need to provide a collection of sets that is (1) large enough to model everything one would normally want to model in a computer program, and (2) small enough that it does not exhaust all the sets in $\mathbb{S}et$. Henceforth, let the set $\mathbb{S}$et (pronounced "fat set") denote the collection which we draw from to model all program objects in our language. Certainly $\mathbb{S}$et must have sufficient modeling power for all programming objects. Using $\mathbb{S}et$ as $\mathbb{S}$et, however, would not leave a specifier any sets to describe the language with. For example, one would require sets that were *larger* than $\mathbb{S}$et when writing the specifications for a RESOLVE compiler that was written in RESOLVE.

To provide sufficient models for programming objects, $\mathbb{S}$et must be closed under the basic type operations of the language. Assume A and B are types that are modeled by sets in $\mathbb{S}$et. Then any type expression that can be derived from A and B must also be contained in $\mathbb{S}$et. RESOLVE currently permits the operators $\times$, $\rightarrow$, and $\mathcal{P}$ in type expressions. Therefore, if A and B are elements of $\mathbb{S}$et, $\mathrm{A} \times \mathrm{B}$, $\mathrm{A} \rightarrow \mathrm{B}$, and $\mathcal{P}(\mathrm{A})$ must also be elements of $\mathbb{S}$et.

Echelons are closed under these basic operations. The properties of echelons include closure under membership, pairing, unions and power sets, which means they are also closed under operators $\times$ and $\rightarrow$.[5] We can define an echelon operation on A, $\mathcal{E}(\mathrm{A})$, to be the smallest echelon that contains A. If we take $\mathbb{E}_0 = \phi$, then $\mathbb{E}_1 = \mathcal{E}(\mathbb{E}_0)$ contains the sets $\phi, \mathcal{P}(\phi), \mathcal{P}(\mathcal{P}(\phi)), \ldots$, which are traditionally used to model the natural numbers. It can be shown that $\mathbb{E}_1$ is only countably infinite, so it will not be large enough for real world models. $\mathbb{E}_2 = \mathcal{E}(\mathbb{E}_1)$, however, does provide sufficient sets. It contains models for $\mathbb{N}, \mathbb{R}, \mathcal{P}(\mathbb{R}), \mathbb{R} \times \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$, etc.

If $\mathbb{S}$et is at least $\mathbb{E}_2$ we know it has sufficient modeling power for all ordinary programs. In RESOLVE, $\mathbb{S}$et is generally assumed to be $\mathbb{E}_2$, but whether it is $\mathbb{E}_2$, or $\mathbb{E}_3$, or $\mathbb{E}_{100}$, the important fact is that a specifier still has access to $\mathcal{E}(\mathbb{S}\mathrm{et})$ to describe the language itself. A rigorous treatment of echelons can be found in [10]. The objective of this summary is only to present enough information to give an idea of their significance for program specification.

3.2 Primitive Types

If sets are the building blocks of the RESOLVE language, then primitive types are the cornerstones on which the other blocks rest. Declarations of primitive types take the form:

$$\begin{aligned}
\mathrm{T}_0 &: \ \mathbb{S}\mathrm{et} \\
\mathrm{T}_1 &: \ \mathbb{S}\mathrm{et} \rightarrow \mathbb{S}\mathrm{et} \\
\mathrm{T}_2 &: \ \mathbb{S}\mathrm{et} \times \mathbb{S}\mathrm{et} \rightarrow \mathbb{S}\mathrm{et} \\
\mathrm{T}_3 &: \ \mathbb{S}\mathrm{et} \times \mathbb{S}\mathrm{et} \times \mathbb{S}\mathrm{et} \rightarrow \mathbb{S}\mathrm{et} \\
&\ \vdots
\end{aligned}$$

[5]Assuming appropriate definitions of $\times$ and $\rightarrow$, we can show that $\mathrm{A} \times \mathrm{B} \subseteq \mathcal{P}(\mathcal{P}(\bigcup\{\mathrm{A}, \mathrm{B}\}))$, and $A \rightarrow B \subseteq \mathcal{P}(A \times B)$.

Most often only the first two will be seen. The first type, T_0, is a simple type, while the remaining types are composite. Like all composite types, primitive composite types cannot be used in isolation—they must have parameters. For example, if Str: $\mathbb{S}$et $\rightarrow$ $\mathbb{S}$et, then one cannot declare x: Str, but one can declare an x: Str(Gamma), where Gamma: $\mathbb{S}$et.

A primitive type, like every other object in RESOLVE, is a set. Abstractly, a type is distinguished from other sets of the same cardinality by its properties. For example, it can be shown that the sets $\mathbb{N}$ and $\mathbb{Z}$ have the same cardinality, but the set $\mathbb{N}$ is not closed under subtraction, while the set $\mathbb{Z}$ is. Primitive types in RESOLVE are introduced via two constructs. First, a definition spells out the properties of the type:

```
Def Is_Natural_Number_Like(N: $\mathbb{S}$et, 0: N,
                                   suc: N $\rightarrow$ N): $\mathbb{B}$ =
    (*P1*) $\forall$n : N, suc(n) $\neq$ 0 and
    (*P2*) Is_Injective(suc) and
    (*P3*) $\forall$S : $\mathcal{P}$(N),
               if 0 $\in$ S $\wedge \forall$n : N, n $\in$ S $\Rightarrow$ suc(n) $\in$ S
               then S = N;
```

Then an assumption introduces a set that satisfies that definition:

Assumption Is_Natural_Number_Like($\mathbb{N}$, 0, suc);

The properties in the definition (P1–P3) mirror the axioms one would normally see in an axiomatic description of the natural numbers. The approach of using definitions to describe the properties of a type simplifies the semantics of the language—we do not have to concern ourselves with special syntax and semantics for signatures and axioms. In RESOLVE, definitions are used to introduce all mathematical objects, whether they are constants, variables, functions, or types. The above assumption indicates that the set $\mathbb{N}$ (together with sets 0 and suc) is any arbitrary model of the natural numbers. This enforces abstractness because the natural numbers are not identified with one particular representation.

3.3 Objects as Sets

Every programming object in the RESOLVE language can be modeled by a set contained in $\mathbb{S}$et. That is, any variable, function, or type that occurs in a programming context must lie within $\mathbb{S}$et. Though math objects are exempt from this restriction, most math objects seen in programs will also be in $\mathbb{S}$et because they are typically used to describe program objects. When we want to describe complex software like compilers and verifiers, our specifications will draw on objects that lie outside of $\mathbb{S}$et.

Simple primitive types are directly contained in $\mathbb{S}$et, and composite primitive types always take parameters, which also puts them in $\mathbb{S}$et. The set operators that we are permitted to use in math type expressions ($\times$, $\rightarrow$, and $\mathcal{P}$) are all closed under echelons. Since all math types that are used to model program types are constructed by applying composite types and set operators to other math types, all such math types are in $\mathbb{S}$et.

All program types have a mathematical model, which is a math type expression. The declaration:

Type Family Stack **is modeled by** Str(Entry);

is the text equivalent to:

Type Family Stack $\subseteq$ Str(Entry);

The subset operator is used instead of the equal operator because of constraints on the model. For an example, see Figure 1.

All programming objects in RESOLVE belong to some type, as indicated by the type membership operator (:). Since all types are modeled by sets in $\mathbb{S}$et, the type membership operator can be replaced with set membership ($\in$) to describe the mathematical relationship between an object and its type. Finally, since $\mathbb{S}$et is closed under membership, all programming objects must be in $\mathbb{S}$et.

4. DISCUSSION TOPICS

During type-checking, a compiler needs to be concerned with a number of questions, such as how to treat subtypes, when to require casts, when to report errors, and when to give warnings. Although these questions must be answered for both program and math types, we focus on how they apply to math types, mainly because of the rich diversity of views on how types should be handled in specification languages, ranging from traditionalists [2, 3, 15] to those whose type systems incorporate theorem provers [11] to those who question the necessity of type systems altogether [6]. Issues involving program subtypes will largely depend on how the language in question handles polymorphism, a topic that merits a separate paper.

The distinction between types and other objects (variables and functions) is quite clear in the programming world: program types are introduced by the keyword **Type**. However, in the math world all objects—variables, functions, and types—are introduced by the keyword **Definition** or through quantifiers in expressions. This uniformity is intentional, since all objects are sets, but it forces specifiers and compilers to rely on other cues to tell them which mathematical objects can be used as types. Examples based on subtypes are discussed in this section.

If an object T is declared to be of type $\mathcal{P}$(A) where A is a type, then T is also a type, and we say that T is a subtype of A. Permitting such declarations requires the language to have reasonable semantics for handling the relationship between the type T being declared and the type A being used in the declaration. Consider the definition:

Definition $\mathrm{Even} : \mathcal{P}(\mathbb{N}) = \{\mathrm{n} : \mathbb{N} \mid \mathrm{n}\ \mathbf{mod}\ 2 = 0\}$;

It is reasonable to want to declare objects of type Even and add them together using the + operator defined in Natural_Number_Theory (the theory introducing $\mathbb{N}$). The type of the result would be $\mathbb{N}$, so a specifier could write:

$$\forall \mathrm{e1}, \mathrm{e2} : \mathrm{Even}, \exists \mathrm{n} : \mathbb{N} \ni 2 \cdot \mathrm{n} = \mathrm{e1} + \mathrm{e2};$$

To analyze this expression, a compiler needs to know that Even is a subtype of $\mathbb{N}$, and it must have an algorithm that determines which + operator to use if there is more than one choice. This can become non-trivial, and as a rule, if something is complex for the compiler, it is also conceptually complex for the programmer or specifier. One way to simplify things is to require explicit type casting, so the above expression would produce an error if there were no + operator defined that took two objects of type Even. To use the + from natural number theory, a specifier might be forced to write:

$$\forall \mathrm{e1}, \mathrm{e2} : \mathrm{Even}, \exists \mathrm{n} : \mathbb{N} \ni 2 \cdot \mathrm{n} = (\mathbb{N})\mathrm{e1} + (\mathbb{N})\mathrm{e2};$$

This makes the expression harder to write since the specifier must do the work that the compiler would have done to decide which + should be be used. In RESOLVE, where emphasis is on qualities such as reuse and understandability, readability usually takes precedence over writability. In this example, there is an argument for both sides in terms of readability. If the + operator is overloaded in an unconventional way, explicit casting may clarify things; if the + operator is not overloaded at all, explicit casting simply adds unnecessary clutter to the expression.

In some programming languages, casting to a parent type is implicit, but casting to a subtype must be explicit. An analogous example in the math world might define:

Definition $\mathrm{Vertex} : \mathcal{P}(\mathbb{Z}) = \{\mathrm{z} : \mathbb{Z} \mid 1 \leq \mathrm{z} \leq \mathrm{Max\_Vert}\}$;

Definition $\mathrm{Cost}(\mathrm{G} : \mathrm{Graph}; \mathrm{v1}, \mathrm{v2} : \mathrm{Vertex}) : \mathbb{R}$[6]

If casting to a subtype is required, one must write

$$\begin{aligned}&\forall \mathrm{G} : \mathrm{Graph}, \forall \mathrm{z1}, \mathrm{z2} : \mathbb{Z},\\ &\quad \mathrm{Cost}(\mathrm{G}, (\mathrm{Vertex})\mathrm{z1}, (\mathrm{Vertex})\mathrm{z2}) \leq 4.7;\end{aligned} \tag{1}$$

instead of

$$\forall \mathrm{G} : \mathrm{Graph}, \forall \mathrm{z1}, \mathrm{z2} : \mathbb{Z}, \mathrm{Cost}(\mathrm{G}, \mathrm{z1}, \mathrm{z2}) \leq 4.7; \tag{2}$$

This may seem quite reasonable for a programmer, but some specifiers may consider the following expression perfectly reasonable:

$$\begin{aligned}&\forall \mathrm{G} : \mathrm{Graph}, \forall \mathrm{z1}, \mathrm{z2} : \mathbb{Z},\ \mathbf{if}\ \mathrm{z1}, \mathrm{z2} \in \mathrm{Vertex}\\ &\quad \mathbf{then}\ \mathrm{Cost}(\mathrm{G}, \mathrm{z1}, \mathrm{z2}) \leq 4.7;\end{aligned} \tag{3}$$

There is nothing wrong with expression (3) as a mathematical formula, and it is obvious that the Cost function is defined for all z1, z2 $\in$ Vertex. But if we insist that the compiler must report a type error for expression (2), then we must insist that it does the same for expression (3). There may be merit in exploring ways that allow the specifier more flexibility in writing expressions while still insisting that he provide sufficient clues to the compiler of his intentions. For example, we might allow:

$$\begin{aligned}&\forall \mathrm{G} : \mathrm{Graph}, \forall \mathrm{z1}, \mathrm{z2} : \mathbb{Z},\ \mathbf{if}\ \mathrm{z1}, \mathrm{z2} : \mathrm{Vertex}\\ &\quad \mathbf{then}\ \mathrm{Cost}(\mathrm{G}, \mathrm{z1}, \mathrm{z2}) \leq 4.7;\end{aligned} \tag{4}$$

[6]We can imagine that the Cost function indicates the expense of traveling from v1 to v2 in graph G.

Expression (4) replaces the set membership operator (∈) of expression (3) with a type membership operator (:). This could indicate to the compiler that z1 and z2 are to be treated as belonging to type Vertex for the remainder of the expression scope. Unfortunately, we would have to develop another mechanisms for the case where the **if** part of expression (4) were in a precondition and the **then** part of the expression were in a postcondition. If we introduce too many distinct mechanisms for handling a conceptually similar situation we run the risk of significantly complicating the language.

All of the questions that arise with subtypes due to the power set operator may occur with primitive types as well. It is reasonable to think of $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ as distinct types—after all, their algebraic structures are different. It is also reasonable to want to treat $\mathbb{N}$ as a subset of $\mathbb{Z}$ and $\mathbb{Z}$ as a subset of $\mathbb{R}$. If these relationships between primitive types are desired, a mechanism different from the one for subtypes must be provided that allows the compiler to treat them as such.

5. RELATED WORK

Many examples of integrated languages exist, though the degree of integration varies widely. Eiffel [9] is essentially a programming language with a few specification features built in. Like RESOLVE, it was created independently of any preexisting programming language; unlike RESOLVE, its specification features are limited—it does not include a complete formal specification language (see p. 400 of [9]). JML (Java Modeling Language) [7] is a behavioral specification language that was created for Java. Used together, JML and Java form an integrated language. Unlike Eiffel, JML provides models for its programming objects. Mathematical expressions in both Eiffel and JML are designed to look similar to programming expressions. Accordingly, they will also type-check similarly. Recall that RESOLVE type-checks mathematical expressions by structure and programming expressions by name. RESOLVE/C++ [5] applies the RESOLVE framework and discipline to the C++ programming language. It uses the specification portion of the RESOLVE language for reasoning. Integrated languages formed by combining a preexisting specification language with a preexisting programming language will type-check mathematical expressions in accordance with the rules of the specification language and will type-check programming expressions in accordance with the rules of the programming language.

Most practical specification languages allow some form of subtyping [2, 7, 15]. The PVS verification system [11] permits downcasting to predicate subtypes by generating a proof obligation when a type is detected in a place where its subtype is expected. Problems similar to those presented in section 4 cause Lamport to question whether specification languages should be typed at all [6]. Topics relating to program subtypes include behavioral subtypes [8] and matching [1].

6. CONCLUSION

Integrated languages must have an effective method for handling program and math types. Integration requires that a mechanism exist for relating programming and mathematical elements. Type declarations are a natural place to describe this relationship. Practical concerns compel us to treat programming and mathematical objects differently. Program types should match according to their names, and math types should match according to their structure.

The theoretical basis of the specification language will affect which objects can be used as types, and will determine the kinds of models that can be constructed for program objects. We need to distinguish sets that model real world objects in a language from larger sets that are needed to describe compilers and verifiers for that language.

The handling of subtypes in the specification portion of an integrated language offers a series of trade-offs. Systems that allow a specifier greater flexibility in writing expressions run the risk of permitting poor expressions that could be caught quickly with a less tolerant type system.

7. ACKNOWLEDGMENTS

Several people contributed important ideas to this work and made helpful comments about drafts of this article. I would especially like to thank Murali Sitaraman, Bill Ogden and Steven Atkinson.

We also gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-0113181, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or the U.S. Department of Defense.

8. REFERENCES

[1] M. Abadi and L. Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, July 1996.

[2] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, New York, 1993.

[3] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, 1993.

[4] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[5] J. Hollingsworth, L. Blankenship, and B. W. Weide. Experience report: Using RESOLVE/C++ for commercial software. In *Proceedings SIGSOFT FSE.* ACM, November 2000.

[6] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, May 1999.

[7] G. T. Leavens, A. A. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe,

and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12. Kluwer, 1999.

[8] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, Cambridge, United Kingdom, 2000.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, New Jersy, 2nd edition, 1997.

[10] W. F. Ogden. *The Proper Conceptualization of Data Structures*. The Ohio State University, Columbus, OH, 2000.

[11] J. Rushby. Subtypes for specifications. In *Software Engineering - ESEC/FSE '97*, pages 4–19. ACM SIGSOFT, September 1997.

[12] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *Procs. Sixth Int. Conf. on Software Reuse*, pages 266–283. Springer-Verlag, 2000.

[13] M. Sitaraman, W. F. Ogden, G. Kulczycki, J. Krone, and A. Reddy. Performance specification of software components. In *Proceedings of SSR '01*, pages 3–10. ACM/SIGSOFT, May 2001.

[14] M. Sitaraman and B. W. Weide. Component-based software using RESOLVE. *ACM Software Engineering Notes*, 19(4):21–67, 1994.

[15] J. Spivey. *The Z Notation*. Prentice Hall, New York, 1989.