from the Proceedings of the USENIX C++ Conference, Washington, DC, April 22–25, 1991

Representing Semantically Analyzed C++ Code with Reprise

David S. Rosenblum (dsr@research.att.com)

Alexander L. Wolf (alw@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974–2070

Abstract

A prominent stumbling block in the spread of the C++ programming language has been a lack of programming and analysis tools to aid development and maintenance of C++ systems. One way to make the job of tool developers easier and to increase the quality of the tools they create is to factor out the common components of tools and provide the components as easily (re)used building blocks. Those building blocks include lexical, syntactic, and semantic analyzers, tailored database derivers, code annotators and instrumentors, and code generators. From these building blocks, tools such as structure browsers, data-flow analyzers, program/specification verifiers, metrics collectors, compilers, interpreters, and the like can be built more easily and cheaply. We believe that for C++ programming and analysis tools the most primitive building blocks are centered around a common representation of semantically analyzed C++ code.

In this paper we describe such a representation, called Reprise *(REPResentation Including SEmantics)*. The conceptual model underlying Reprise is based on the use of expressions to capture all semantic information about both the C++ language and code written in C++. The expressions can be viewed as forming a directed graph, where there is an explicit connection from each use of an entity to the declaration giving the semantics of that entity. We elaborate on this model, illustrate how various features of C++ are represented, discuss some categories of tools that would create and manipulate Reprise representations, and briefly describe our current implementation. This paper is not intended to provide a complete definition of Reprise. Rather, its purpose is to introduce at a high level the basic approach we are taking in representing C++ code.

1 Introduction

A prominent stumbling block in the spread of the C++ programming language has been a lack of programming and analysis tools to aid development and maintenance of C++ systems. The development of such tools has been, and continues to be, a daunting prospect. One reason for this is that the language itself has been evolving, making it risky to invest much effort in the development of language-specific tools; fortunately, this situation appears to be improving. Another reason is that the language is inherently complex, both in its syntax and semantics. The developer of anything more than the most trivial of tools is faced with the prospect of having to devote a significant portion of their tool—and time—to dealing with this complexity.

One way to make the job of tool developers easier and to increase the quality of the tools they create is to factor out the common components of tools and provide those components as easily (re)used building blocks. We believe that for C++ programming and analysis tools the most primitive building blocks are centered around a common representation of semantically analyzed C++ code.

Consider, for example, the design of the C++ Information Abstractor (CIA++) [9]. CIA++ is a tool that constructs a database of information about the non-local entities in a C++ program. A variety of display and analysis tools make use of this database to provide information to developers. Thus, CIA++ is a building-block tool in the sense that it offers a common service, namely the specialized filtering and structuring of information about C++ code, to a number of other tools. The current version of CIA++ was built by painful modification of *cfront* [1], which is a tool that performs lexical, syntactic, and semantic analysis of C++ code, as well as a translation of the C++ code into C. Relatively simple modifications included the careful removal of all code performing the actual generation of C code. The difficult modifications stemmed from the manner in which the internal data structures and code are organized in *cfront.* Specifically, the collection and representation of semantic information is spread out across the code. Indeed, certain semantic information is "thrown away" at various times during the translation process. While this organization might make sense for translation of C++ to C, it made the design of CIA++ very complex. Furthermore, to maintain compatibility with the "official" *cfront,* updates to the official version must be carefully incorporated into CIA++. Had there instead been available a data structure representing the semantics of C++ code (and, of course, a tool to generate such a data structure) the developers of CIA++ could have written a relatively simple tool to derive CIA++ databases directly from the data structure representation. Thus, the design of CIA++ would have been greatly simplified, the time required to implement it would have been drastically reduced, and the need to track updates to *cfront* would have been avoided.

We have developed Reprise, a representation for semantically analyzed C++ code.[1] Reprise can serve as the basic data structure for the building blocks of a wide variety of tools. Those building blocks include lexical, syntactic, and semantic analyzers, tailored database derivers (e.g., CIA++), code annotators and instrumentors, and code generators. From these building blocks, tools such as structure browsers, data-flow analyzers, program/specification verifiers, metrics collectors, compilers, interpreters, and the like can be built more easily and cheaply.[2] Factoring out the primitive components in this manner would free tool developers to concentrate on the unique, critical aspects of their tools. An additional benefit of this approach is that tools operating on the same C++ code can share the Reprise representation of that code, resulting in a significant savings in both space and time. Given our experience in defining, building, and using Ada programming and analysis tools (e.g., [5, 14, 15, 20]) based on the Diana [7] and Paris [8] representations, we believe that this approach to the development of tools for C++ is a viable one to consider.

We begin in Section 2 by describing the model upon which the representation is based. In Section 3 we sketch, through examples, how C++ code is actually represented in Reprise. In Section 4 we discuss some categories of tools that would create and manipulate Reprise representations. We conclude in Section 5 with a brief description of our current implementation.

Note that this paper is not intended to provide a complete definition of Reprise. Rather, its purpose is to introduce at a high level the basic approach we are taking in representing C++ code.

[1] Reprise is an acronym for *REPResentation Including SEmantics.* We intend this name to evoke a feeling of reuse of representations, as in the musical term *reprise,* "a repetition of a phrase or verse" [11].

[2] Whether something is viewed as a tool or as a building block for tools is, of course, a matter of perspective. For all intents and purposes, a building block is a kind of tool, so we do not distinguish between building blocks and tools in the remainder of this paper.

2 The Representation Model

As mentioned above, we imagine that a representation for semantically analyzed C++ code could be used profitably by a wide variety of tools. The accommodation of such variety requires that the representation exhibit the following characteristics:

- *Primacy of semantics.* The form of the representation must be driven by the semantics of the language constructs, not by their syntax, since it is primarily at the semantic level that sophisticated tools need to manipulate information about C++ code.[3]

- *Regularity of form.* The manipulations needed for basic processing of the representation, such as traversal, must be straightforward to understand and implement. Regularity of form reduces the need for tools to contain complicated, special-case code.

- *Minimization of speciality.* Different tools may want to treat portions of C++ code differently. These necessary and desirable idiosyncrasies, however, must not be embedded in the form of the representation. Rather, they should be reflected in the tools themselves. We cannot expect to be able to anticipate the needs of all tools and, moreover, those needs may be in conflict.

- *Evolvability of representation.* Although the language definition is stabilizing, there continue to be proposals for changes (e.g., for templates [16] and exception handling [10]). It is important, therefore, that the representation can be easily evolved along with the language. To facilitate this, the representation must capture more than just the surface-level, user-visible semantics. It must also effectively capture the basic fabric of the language—that is, the semantics of how the language is put together.

- *Uniformity of representation.* The basic fabric of C++ must be represented in the same way as user-written code. Indeed, cognizance that a particular entity is primitive (i.e., a component of the basic fabric) or not is something that should be left up to individual tools. Uniformity of representation, like regularity of form, reduces the need for special-case code in tools. Moreover, it allows tools to be constructed in such a way that they can more easily evolve along with the language and representation.

These characteristics form recurring themes throughout the design of REPRISE.

The conceptual model underlying REPRISE can be viewed from two equivalent perspectives. The first is as an expression language in which all semantic information about both the C++ language and code written in C++ is uniformly represented as the application of operators to arguments. For example, the expression `F != 0`, where `F` is a pointer variable, is represented as an application of the language-defined inequality operator `!=` to the object `F` and the literal `0`. A more interesting example is a representation for a construct like the *if-statement,* where the language-defined operator `if` is applied to two arguments, one an expression representing a condition and the other an expression representing the statement to execute if the condition is true. Even declarations are represented as expressions, as discussed in Section 3.

A second way of viewing REPRISE, and the one that we tend to use most often, is as a directed graph. Unlike a traditional abstract syntax tree, a REPRISE graph explicitly captures semantic information by connecting, through edges, entity uses with entity declarations. To do so, a REPRISE graph, which could well be called an "abstract semantics graph", employs two kinds of nodes and two kinds of directed edges. The first kind of node is used to represent expressions (i.e., applications of operators to arguments) and the second kind is used to represent literals (e.g., identifiers, numbers, or strings). Expression nodes have one or more children. The first child is always an expression representing the declaration, and hence semantics, of the operator being applied, while the other children are the arguments to the operator. The two kinds of edges are used to distinguish between the following two situations: *i*) an expression node having a child that is the result of a previously evaluated expression, and *ii*) an expression node having a child that is an expression to be evaluated

[3] Note that this characteristic can lead to easy accommodation of graphics-based tools (i.e., those that present C++ code to users through an iconic, rather than textual, syntax).

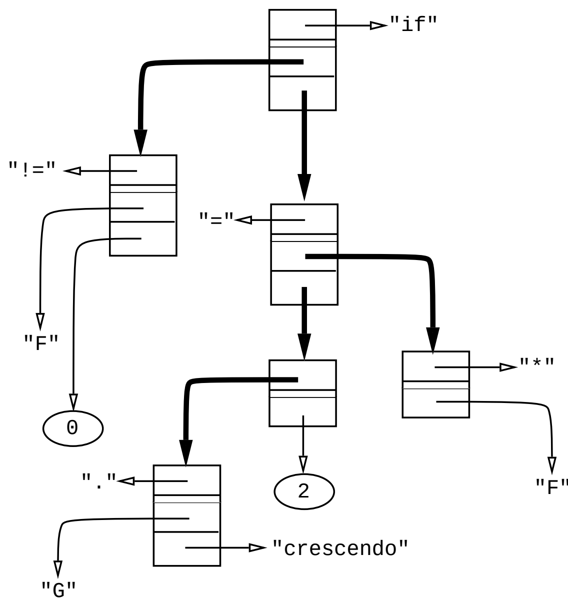

Figure 1: REPRISE Representation of an if-statement.

as part of the parent expression's evaluation. We call the first kind of edge a *reference edge,* since the parent expression is actually referring to a previously determined result, and call the second kind an *evaluation edge,* since the child expression is evaluated as part of the application of the parent's operator. Reference edges are also used to refer to children that are literal nodes. The need to distinguish edges is a rather subtle issue explored further in Section 3. Fortunately, while the distinction must be evident in the representation, it can in practice be ignored by most tools.

Figure 1 depicts a portion of a REPRISE graph representing the code fragment

```
if (F != 0) *F = G.crescendo(2);
```

where rectangles are expression nodes, ovals are literal nodes, dark arrows are evaluation edges, and light arrows are reference edges. The connections between this code fragment and the declarations for entities `if`, `!=`, `*`, `=`, `.`, `crescendo`, `F`, and `G` are implied by the pictorial abbreviation →`"entity"`. (Many of the abbreviations appearing in this figure are expanded in Figure 3.) Although it is not evident from the depiction, all occurrences of the same abbreviation in this figure denote the same entity declaration. Thus, both edges →`"F"` actually terminate at the same node, the root of the subgraph representing the declaration of `F`. It is this kind of sharing that makes REPRISE a non-tree graph. Indeed, a REPRISE graph may contain cycles.

The connections between uses and declarations drawn by edges are what capture the semantics of the code being represented. For instance, looking at the expression node in Figure 1 that represents the selection of member `crescendo` from object `G`, we see that it is an application of the language-defined "dot" operator (i.e., the operator whose declaration is at the other end of the reference edge[4]) to a particular object (the one whose declaration is at the other end of the reference edge) and an expression (at the other end of the evaluation edge). A more extensive example is given in the next section.

While a REPRISE graph clearly is not an abstract syntax tree, one easily can see where the traditional abstract syntax tree resides, as a subgraph, within it. The existence of this subgraph is important, since tree-based algorithms are generally more efficient than their graph-based counter-

[4] A reference edge is used here because functions in C++ are first declared (the declaration expression is evaluated) and then applied (referenced) in some number of other places, such as this one, in the code.

parts; the abstract syntax tree subgraph can be used where appropriate in manipulating a REPRISE representation.

Another subgraph of interest in a REPRISE representation is what we refer to as the *core graph.* The core graph comprises all the nodes and edges used to represent the semantics of C++.[5] Thus, the core graph is totally self contained; no edges leave the core graph. Included in the core graph are representations of such things as the declarations of the operators **`if`** and **`switch`**, as well as declarations for the types `int` and **`float`**. Also included are even more primitive operators and types, such as **`%list`** and **`%numeric`**,[6] that are not explicit in any user-written code, but are employed both in the representation of the primitive operators and types themselves and in the representation of user-written code. Conceptually, the core graph forms part of every REPRISE representation. Because it is self contained, however, the core graph can be physically shared among those representations.

The core graph provides one view of the C++ primitive semantics. Taking the expression-language perspective, the primitive semantics can be viewed as a set of expressions declaring the primitive types and operators. In other words, C++ can be described as a particular set of declared entities, where user-written code makes use of those declared entities. The advantage of this perspective is that it becomes evident what the relationships, and "non-relationships", are among the primitive entities. Moreover, additions to the language, such as those proposed for exception handling, amount to the declaration of some additional entities.

Let us relate the model underlying REPRISE to the characteristics listed at the beginning of this section. The primacy of semantics is reflected in the explicit connection in the representation from each use of an entity to the declaration giving the semantics of that entity. Indeed, it is these semantic connections that make the representation much more than a typical abstract syntax tree. Representing all aspects of C++ code as expressions and employing only two kinds of nodes and two kinds of edges leads to both a very regular form for representations and a minimum of speciality. Nevertheless, accommodating the special needs of specific tools, such as keeping track of exercised branches and statements for test-coverage analysis, is straightforward, as discussed in Section 4. Finally, the REPRISE core graph captures the complete primitive semantics of C++ and, moreover, does so in the same form as user-written code. Having the primitive semantics in this form is useful in evolving the representation to accommodate new language features, since it is a particularly malleable data structure.

3 Representing C++ Code with Reprise

In this section we describe how C++ code is represented as REPRISE graphs. The terminology we use generally follows that of the C++ reference manual [6]. For purposes of illustration, we use the REPRISE representation of the C+ program of Figure 2, a simple example involving a class called `ical`. The REPRISE representation of this program is shown in Figure 3, except that core graph entities are represented by the pictorial abbreviation →`"entity"`, as before. Note that the source program of Figure 2 includes the if-statement whose REPRISE representation is depicted in Figure 1; thus, Figure 3 also illustrates how the representation of the if-statement fits into the larger context of a complete C++ program.

The entities appearing in C++ user-written code fall into three broad categories: types, declarations, and statements. We illustrate each of these categories with representative examples extracted from Figure 3 and then conclude with a discussion of some of the conventions we use in representing C++ code with REPRISE.

3.1 Representation of Types

There are three aspects to the representation of types in a REPRISE graph: the hierarchy of predefined C++ types, a collection of operators called *type constructors* for representing the formation of user-

[5] Of course, the most primitive semantics, namely operator application and argument evaluation, are not explicitly represented in the graph, but are assumed to be "understood" by all tools.

[6] We prepend the character "`%`" to the names of primitive entities not available for use in user-written C++ code.

```
class ical {
private:
      int p, f;
public:
      ical crescendo(int const c);
      ical() { p = f = 1; }
};

void main()
{
      ical* F = 0;
      ical G;
      if (F != 0) *F = G.crescendo(2);
}
```

Figure 2: A Sample C++ Program.

defined types, and a collection of operators called *type modifiers* for representing additional attributes of types.

Figure 4 depicts the hierarchy of predefined types in C++. As shown in the figure, this hierarchy is a subtype relationship. The subtype relationship is defined inductively in terms of the functions that are defined for each type T. In particular, the functions that are *applicable* to objects of type T include *i*) zero or more functions that are defined explicitly for T, plus *ii*) the functions applicable to object's of T's supertype. In addition, the language defines certain implicit type promotions and conversions that allow the functions for one type to be applied to objects of another type. In other words, if a function applied to an object of type T is not defined for type T, then either the function must be defined for some supertype of T, or else there must be an implicit type promotion or conversion defined by the language from type T to type T' such that the function is defined for type T'.

The fundamental types of C++ (`char`, `int`, etc.) appear at the "leaves" of the subtype hierarchy and are shown in boldface in Figure 4. The remainder of the hierarchy comprises several *meta-types* (denoted by a leading "`%`"), which are never used explicitly within user-written code. Figure 3 illustrates references to the fundamental types `int` and `void`, as well as references to the meta-types `%func`, `%class`, and `%pointer`.

The utility of the subtype hierarchy can be appreciated by considering the fundamental type `int`. The language predefines several functions for `int`, namely the arithmetic operators (unary and binary `+` and `-`, `*`, `/`, and `%`), bitwise operators (`~`, `&`, `|`, and `^`), shift operators (`<<` and `>>`), and relational operators (`<`, `>`, `<=`, `>=`, `==`, and `!=`). As shown in Figure 4, `int` is a subtype of `%numeric`. The language predefines several functions for `%numeric`, namely the ternary conditional operator (`?:`), increment operators (`++`), decrement operators (`--`), and logical operators (`!`, `&&`, and `||`). Thus, because `int` is a subtype of `%numeric`, all of the operators defined for `%numeric` are applicable to objects of type `int`.

A type constructor represents the formation of a user-defined type from one or more existing "ingredient" types,[7] while the type modifiers qualify a type as being either a constant, volatile, or reference type. Each user-defined type is a subtype of one of the meta-types shown in boxes in Figure 4. In order to represent the subtyping semantics described above, one argument to each type constructor is a reference to the type's supertype in the subtype hierarchy. The other arguments to a type constructor represent other attributes of the constructed type, including the ingredient type(s) from which it is constructed (such as the base types of a class). The sole argument to a type modifier is a reference to the modified type. Figure 3 illustrates the use of the type constructors `class`, `%function`, and `*`, as well as the type modifier `const`.

The operator `class` is used to represent the definition of a class, structure, or union. All non-

[7] User-defined types are referred to as "derived types" in Section 3.6.2 of the C++ reference manual [6].

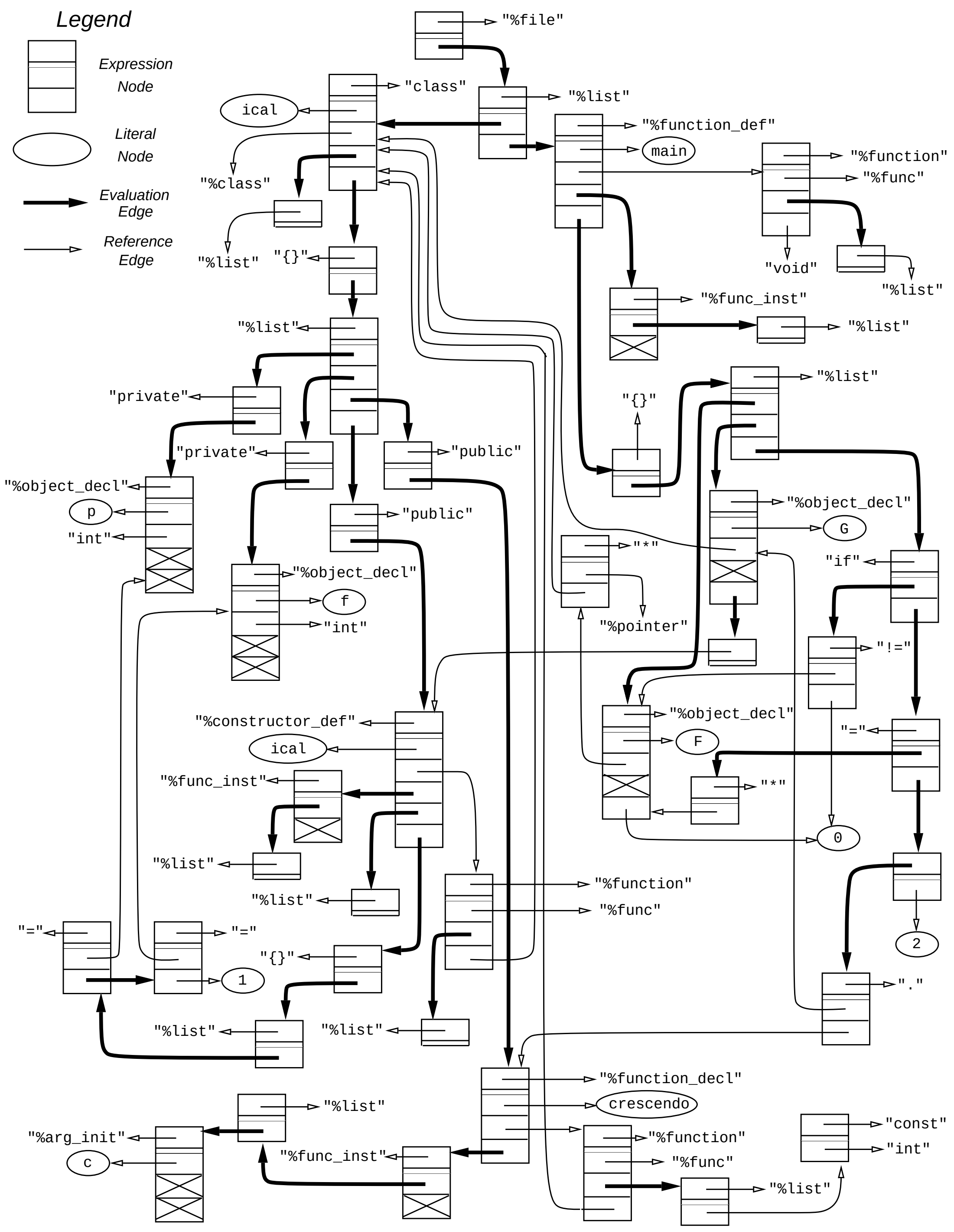

Figure 3: REPRISE Representation of the C++ Program of Figure 2.

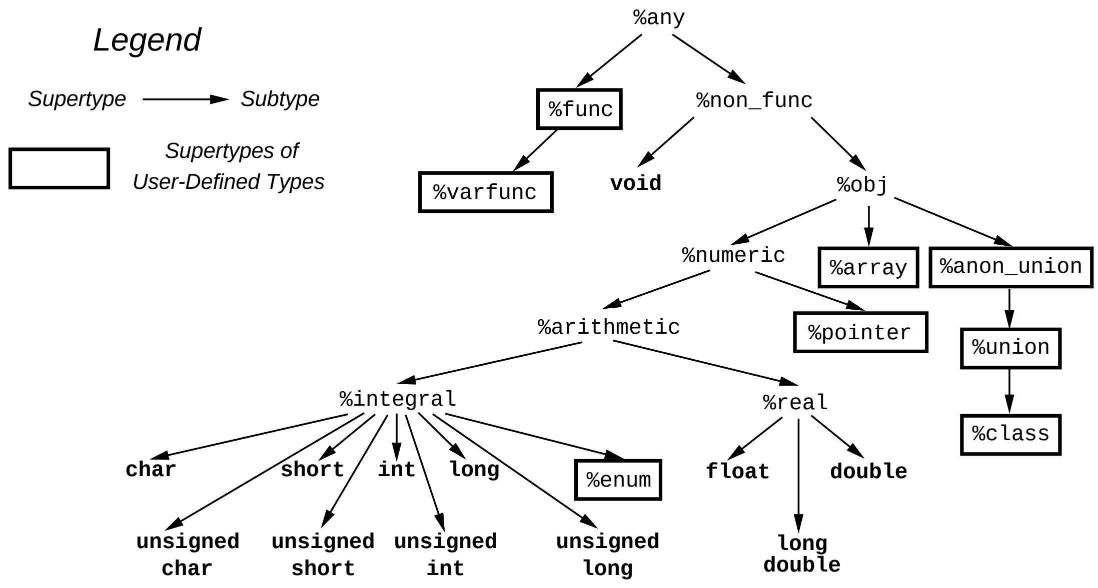

Figure 4: Subtype Hierarchy of Predefined C++ Types.

union classes are subtypes of the meta-type `%class`. Anonymous unions are subtypes of the meta-type `%anon_union`, while all other unions are subtypes of the meta-type `%union` (see Figure 4). Note that from the perspective of the subtype relationship, a structure is semantically the same as a class and is thus a subtype of the meta-type `%class`. Non-union classes are further related according to an inheritance relationship that is specified explicitly in the user-written code, using the inheritance syntax of C++. The semantics of the inheritance relationship is quite different from the semantics of the subtype relationship, and hence the two relationships are represented in different ways in a REPRISE graph. In particular, given a class C and a class D derived from C, the inheritance relationship between the two classes is represented by a reference edge from the representation of D to the representation of C. On the other hand, both classes are subtypes of `%class`, and therefore the representations of the classes are connected by reference edges to the representation of `%class`. If there also happens to be a subtype relationship between C and D, this fact can be determined from their inheritance relationship and from their definitions. The differences in the semantics of the inheritance and subtype relationships are discussed in detail by Moss and Wolf [12].

Figure 5 illustrates the use of the operator `class`; it contains the portion of the graph of Figure 3 devoted to the representation of class `ical`. As shown in the figure, `class` takes a name as its first argument, the appropriate supertype as its second argument, a list of base classes as its third argument, and a list of member declarations as its fourth argument. Both lists are represented by an application of the operator `%list`, which is defined in the core graph and can take a variable number of arguments. Because class `ical` has no base classes, the second argument to this application of `class` is an empty list.

The operator `%function` is used to represent the type of a function in the representation of function declarations, function definitions, and pointer-to-function types. Figure 6 illustrates the use of the operator `%function`; it contains the portion of Figure 3 devoted to representing the type of member function `crescendo` of class `ical`. Figure 6 also illustrates the use of the type modifier `const`, which simply qualifies its argument as being a constant type. The first argument to `%function` is a reference to the supertype of the function type, the second argument is a list of argument types, and the third argument is a reference to the return type. The supertype of a function type having a fixed number of arguments is the meta-type `%func`, while the supertype of a function type having a variable number of arguments is the meta-type `%varfunc`. Note that

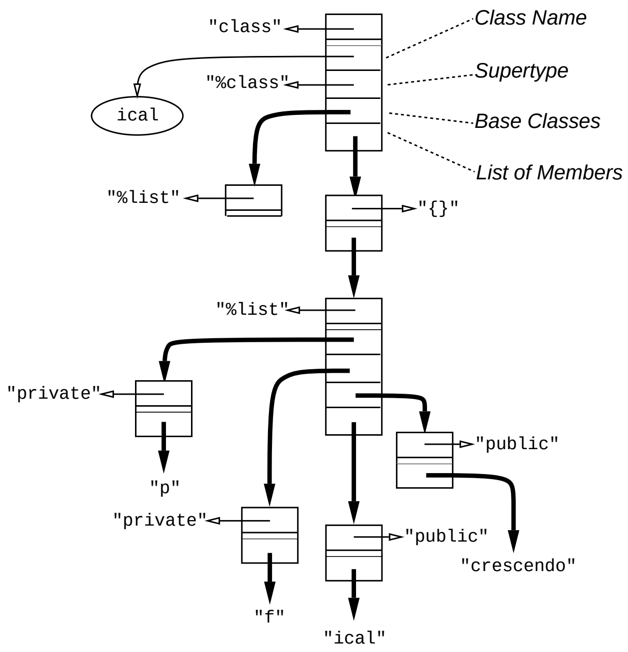

Figure 5: REPRISE Representation of Class `ical`, Extracted from Figure 3.

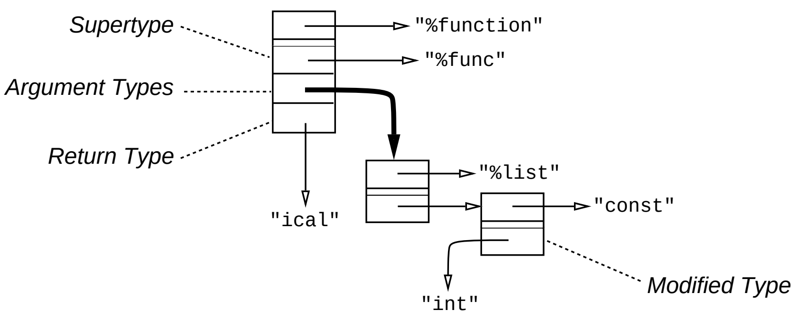

Figure 6: REPRISE Representation of the Type of Function `crescendo`, Extracted from Figure 3.

argument names and argument initializers are not semantically part of the type of a function; for instance, two functions that have the same set of argument and return types, but different sets of argument names and initializers, could both be assigned to a variable of the appropriate pointer-to-function type (which may have been declared with still another set of names and initializers). Therefore, argument names and initializers are associated with the representation of each declaration whose type involves a function type (e.g., a function, or a variable that is a pointer to a function), instead of with the representation of the function type itself (see Section 3.2).

Two additional things should be noted about the representation of types in a REPRISE graph. First, supertypes and ingredient types are existing types from which new types are built; thus, when supertypes and ingredient types are specified as arguments to a type constructor, they are specified with reference edges. Second, when newly defined components of a type (such as a list of class members or function argument types) are specified as arguments to a type constructor, they are specified with evaluation edges.

3.2 Representation of Declarations

C++ declarations are represented by expressions involving operators called *declaration constructors*. A declaration constructor represents the declaration of a variable, function, or typedef. Figure 3 illustrates the use of the declaration constructor for variables (`%object_decl`) and three of the declaration constructors for functions (`%function_decl`, `%function_def`, and `%constructor_def`). Note that class members are represented simply as variables or functions that are defined within classes. In addition to the declaration constructors, several *declaration modifiers* are used to represent various attributes of declarations, such as access specifiers (e.g., `private`), storage class specifiers (e.g., `static`), function specifiers (e.g., `friend`), and linkage specifications (e.g., `extern "C"`).

All declarations in C++ code have an associated *scope* within which they are visible. Scopes are represented in a REPRISE graph by several operators that are defined in the core graph. In particular, a file scope is delimited by an application of the operator `%file`, as shown in Figure 3. The scope of a function argument or statement label is delimited by an application of one of the declaration constructors for functions. The scope of a variable declared in a for-loop header is delimited by an application of the operator `for`. All other scopes are delimited by the operator {},[8] which is used to denote the scope of a declaration in a compound statement (including the outermost scope of a function body) and the scope of a class member.[9]

3.2.1 Variable Declarations

A variable declaration is represented by an application of the operator `%object_decl`. Figure 7 depicts the portion of the graph of Figure 3 devoted to the representation of data member `p` of class `ical`. The first argument to `%object_decl` is the name of the declared object. The second argument is a reference to the type of the object. The third argument is the list of name/initializer pairs for the type; in this case, the third argument is null since the type does not involve a function type. The fourth argument is an initializer for the object, which in Figure 7 is null because no initializer is specified for `p`.

The `%object_decl` expression of Figure 7 by itself simply represents the declaration of a variable called `p`. However, since `p` is a member of class `ical`, the `%object_decl` expression appears as the argument to an instance of the declaration modifier `private`, which represents the fact that `p` is a private class member. In addition to the declaration modifier `private`, there are also declaration modifiers `public` (see Figure 3) and `protected` for representing access specifiers of class members.

The representation of the declaration of `G` in Figure 3 illustrates how initialization of class objects is represented. Even though no initializer is specified explicitly for this variable in the program of Figure 2, semantically it is initialized by the constructor that is defined for class `ical`. Thus, the fourth argument to the `%object_decl` expression is an expression representing a call to this constructor.

[8] The operator {} is pronounced "Curly".

[9] Note that the access specifiers `protected` and `public` in effect *extend* the scope of a class member.

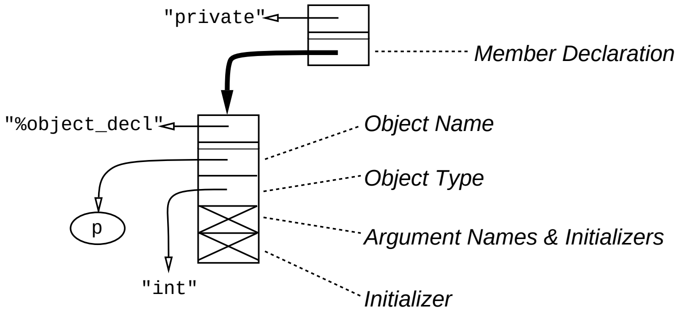

Figure 7: REPRISE Representation of Data Member `p` of Class `ical`, Extracted from Figure 3.

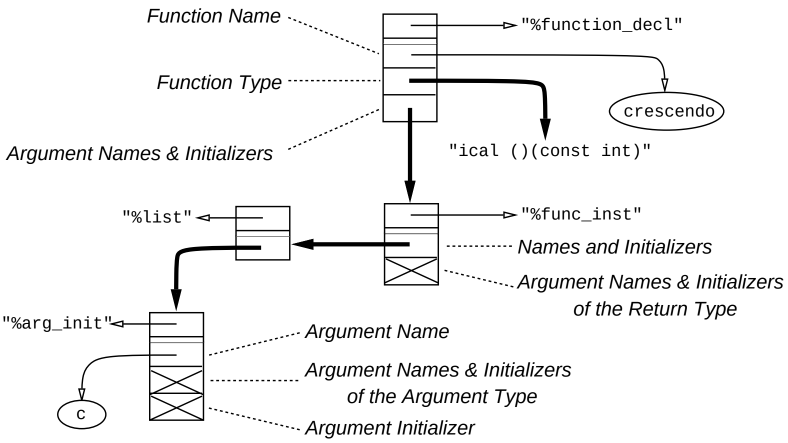

Figure 8: REPRISE Representation of Member Function `crescendo`, Extracted from Figure 3.

As is done in the representation of types, reference edges and evaluation edges are used to distinguish, respectively, between the existing entities in a REPRISE graph upon which a declaration depends (such as the type of the declared entity) and those entities that are defined specifically for a declaration (such as the initializer of an object, or the argument name/initializer pairs and body of a function).

3.2.2 Function Declarations and Definitions

A function is represented by an application of either the operator `%function_decl` or the operator `%function_def`. The former is used to represent functions declared with a header but no body, while the latter is used to represent functions that are declared with their bodies. The only difference between the two is that the latter has an extra argument for representing the function body.

Figure 8 contains the portion of the graph of Figure 3 devoted to the representation of member function `crescendo` of class `ical` and illustrates the use of the operator `%function_decl`. The first argument to `%function_decl` is the name of the declared function. The second argument is a reference to the type of the function; the representation of the type of `crescendo` is shown in Figure 6. The third argument represents the name/initializer pairs for this particular use of the function type. We call such a particular use a *function-type instantiation*. As mentioned above, the name/initializer pairs are associated with the function declaration rather than the function type because semantically they are not attributes of the function type.

As shown in the figure, the operator %func_inst is used to represent a function type instantiation. The first argument to %func_inst is the list of name/initializer pairs for the arguments of the function type. The second argument is the %func_inst expression for the return type, which in this case is null since the return type does not instantiate a function type. The operator %arg_init is used to represent each name/initializer pair. The first argument to %arg_init is the name of the argument, the second argument is the %func_inst expression for its type, and the third argument is its initializer. The second and third arguments to %arg_init are both null in this case since c has no initializer and its type does not instantiate a function type.

Class constructors and destructors are special kinds of functions and are represented by applications of the operators %constructor_decl, %constructor_def, %destructor_decl, and %destructor_def. The semantics of the arguments to the operator %constructor_decl are identical to those of %function_decl. The operator %constructor_def is similar to %function_def except that it has an additional argument (its fourth argument) for representing the optional list of initializers for members and base classes; the use of this operator is illustrated in Figure 3, which shows the fourth argument as being an empty list because there are no initializers specified for the constructor of class ical. The operators for representing destructors differ from those for representing "normal" functions only in their lack of an argument for representing function argument/initializer pairs, since destructors do not take arguments.

Overloaded functions are represented using the declaration constructors for functions in the same manner as is illustrated in Figure 3 for functions that are not overloaded. There is nothing special about the representation of an overloaded function other than the fact that multiple function declarations with the same name can can appear in a Reprise graph. However, unlike a simple abstract syntax tree representation of C++ code, a Reprise graph contains no ambiguities as to which function is being called in a reference to an overloaded function, because each such reference is resolved in the form of a semantic connection between the reference and its matching declaration.

C++ defines several functions on its fundamental types, such as the operator + that takes two int arguments and returns an int. The declarations of these predefined functions are represented by %function_decl expressions in the core graph. Certain predefined functions, such as the pointer dereferencing operator *, are polymorphic, since they are defined once for a category of types. A full discussion of the representation and use of polymorphism in Reprise is beyond the scope of this paper.

3.3 Representation of Statements

Representation of C++ statements in Reprise is straightforward. C++ expression statements fit naturally with Reprise's expression-based model and are thus represented simply by nested applications of operators. Each of the remaining kinds of C++ statements is represented by a Reprise operator whose name is the same as that of the statement (if, switch, continue, etc.). Figure 3 illustrates the representation of three kinds of statements—expression statements, if-statements, and compound statements. The operator if is used to represent if-statements and is described in Section 2. The operator {} represents compound statements, simply taking a list of declarations and statements as its sole argument.

3.4 Conventions

Our method of representation adheres to several informal conventions. For example,

- the first argument to each declaration constructor specifies the name of the declared entity;
- the second argument to each declaration constructor specifies the type of the declared entity; and
- an operator argument that is an empty list is represented by a %list expression having zero arguments, rather than by the null value.

These and other such conventions add to the uniformity of the representation, further simplifying the job of implementing Reprise-based programming and analysis tools.

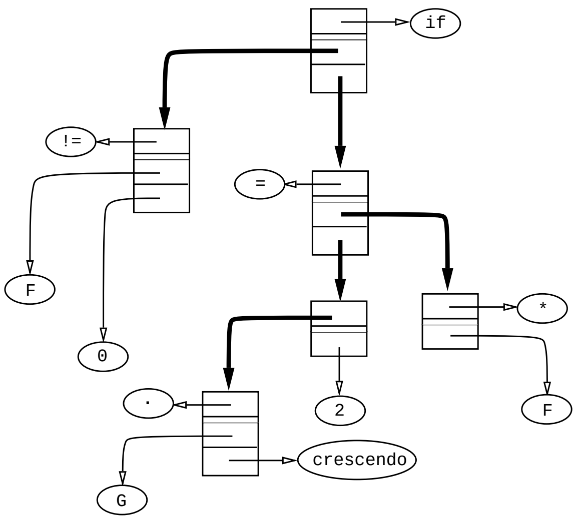

Figure 9: "Pre-semantic" Form of the REPRISE Representation in Figure 1.

4 Reprise and Tools

As discussed in Section 1, the purpose of REPRISE is to serve as a common data structure for programming and analysis tools that depend upon semantic information about C++ code. The previous two sections concentrate on the representation itself. Here we turn attention to the tools that would make use of the representation. There appear to be at least four categories of such tools, each of which is discussed below. For purposes of exposition, the categories are discussed separately, but of course hybrid tools are also possible.

4.1 Generative Tools

Generative tools create REPRISE representations. The most obvious generative tool is what traditionally serves as the front end of a compiler, performing lexical, syntactic, and semantic analysis. The input would be C++ source text and the output would be the REPRISE representation of that source text. One way that this tool could work (in fact, the way that such tools work for the PARIS representation of Ada) is to first generate the abstract syntax tree subgraph through lexical and syntactic analysis, and then modify that subgraph through semantic analysis to capture the semantic connections between entity uses and entity declarations. Figure 9 shows the so-called "pre-semantic" form of the representation shown in Figure 1. In essence, the "post-semantic" form of Figure 1 differs from the pre-semantic form in that certain literal nodes are replaced with semantic connections to the declarations of the referenced entities. Note that as a consequence of the overloading and scoping rules of C++, there is likely to be a many-to-one mapping of literals to declarations in the pre-semantic form. The direct capture of the semantic connections in (post-semantic) REPRISE, however, eliminates any possible ambiguity.

Less traditional generative tools would be C++ semantics-directed editors[10] and graphics-based editors, where the textual syntax of C++ has been replaced with an iconic syntax. While these tools would not take actual C++ source text as input, they would still produce REPRISE representations that could be manipulated by the other categories of tools.

[10] Today's "syntax-directed" editors generally perform semantic checks as well, and thus it is really a disservice to continue calling them syntax-directed editors rather than semantics-directed editors.

4.2 Deriver Tools

Deriver tools create specialized representations of C++ code, using Reprise representations of the code as their input. These specialized representations are tailored to the needs of particular tools or set of tools. A specialized representation might, for instance, contain a subset of the information contained in a Reprise representation or provide a different structure to the information appropriate to a particular kind of processing. CIA++, discussed in the introduction, is an excellent candidate for a deriver tool. Of course, specialized representations do not have to replace the Reprise representations from which they were derived, but can be used in conjunction with those Reprise representations as well. For instance, a tool that creates an index of identifier names, for fast access to declarations in a Reprise graph, can be thought of as a deriver tool.

4.3 Annotation and Instrumentation Tools

Annotation and instrumentation tools "decorate" the Reprise representation of C++ code with specialized information. Examples of tools that produce or make use of such decorations are test-coverage tools, performance analyzers, debuggers, and tools that produce embedded constraint-checking code from formal specifications (e.g., App [13]). A test-coverage tool, for example, might work as follows: Given a piece of code and some test input, the tool tries to make a determination of which branches would be taken in the code and which statements would be executed. For each branch and statement in the code, the tool keeps track of which test input, if any, exercised (i.e., covered) that branch or statement.

The key issue raised by annotation and instrumentation tools is whether specialized, supplementary information, such as test coverage, can be associated with a Reprise representation in an unobtrusive way—that is, without affecting tools not concerned with the information. To some extent, this is a question of how Reprise as an abstraction is actually implemented, since the choice of implementation technique can have a significant impact on this issue [18]. In general, however, we note that Reprise lends itself to an approach in which nodes and edges can be uniquely identified, and that those identities can be used as keys for auxiliary data structures. The values associated with those keys, and hence with nodes and edges, make up the supplementary information relevant to particular tools. Thus, in the case of the test-coverage tool, information about which test inputs exercise which branch or statement can be captured in a separate data structure known only to the test-coverage tool. The connection between that data structure and the C++ code is made by the unique identity of nodes and edges representing branches and statements in a Reprise graph.

An interesting tool to consider in this category is the preprocessor *cpp* [1]. This tool interprets special statements, called *preprocessor directives,* that appear in files containing C++ code. The preprocessor uses the directives to determine what C++ code to pass on to other tools for processing. There are some tools, such as CIA++, that make use of information contained in preprocessor directives, and therefore it is important to be able to retain this information along with the Reprise representation of the actual C++ code. The natural way to do this is to have an auxiliary data structure (perhaps created by a version of *cpp*) that serves this purpose. An important advantage of this approach is that the connections between preprocessor directives and C++ code can be made at a semantic level, not simply a syntactic one.

4.4 "Vanilla" Tools

Generally, tools that do not create Reprise representations, derive specialized representations, or annotate or instrument C++ code with specialized information, use Reprise representations directly and as is. In other words, the Reprise representation contains all the information the tools would need and in a form appropriate to their tasks. Tools that would fall into this category include pretty printers, metrics collectors, data-flow analyzers, inheritance-hierarchy displayers, code generators, and the like.

5 Conclusion

The efficacy of programming and analysis tools using and sharing graph-based representations of semantically analyzed code has already been demonstrated for languages other than C++. The Arcadia environment [17], for example, has a full arsenal of concurrency analysis [2, 21], interface analysis [20], testing [4], and interpretation [22] tools, all based on such representations of Ada and Ada-like code. REPRISE represents an application of this technology to C++ programming environments.

To date, we have implemented the REPRISE data structures as a library of C++ classes. The classes are built on top of a persistence library called *Persi* [19], which supports long-term storage of C++ objects and shared concurrent access to those objects. We have built an enhanced version of *cfront* called *rfront*, which generates pre-semantic REPRISE graphs from C++ source code. We have also implemented a variety of tools that manipulate REPRISE representations, including a prototype name resolver that transforms pre-semantic graphs into post-semantic graphs.

We believe that REPRISE provides an excellent mechanism for constructing C++ environments, for simplifying development of C++ programming and analysis tools, and for increasing tool quality. In other words, we see REPRISE increasing the tempo at which C++ programming and analysis tools are developed, and orchestrating an harmonious interaction among them.

Acknowledgements

The conceptual model underlying the design of REPRISE was strongly influenced by IRIS [3]. IRIS is being used in the Arcadia software development environment research project [17] as the basis for representations of code written in a number of languages, including Ada, PIC/ADL [20], and GDL [5].

References

[1] AT&T. *UNIX® System V C++ Language System Release Notes*, release 2.1 edition, 1990. Select Code 307-160.

[2] G.S. Avrunin, L.K. Dillon, and J.C. Wileden. Experiments with constrained expression analysis of concurrent software systems. In *Proceedings of the SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 124–130, Key West, FL, December 1989. ACM SIGSOFT.

[3] D.A. Baker, D.A. Fisher, and J.C. Shultis. The gardens of Iris. Technical report, Incremental Systems Corporation, Pittsburgh, PA, 1988.

[4] L.A. Clarke, D.J. Richardson, and S.J. Zeil. TEAM: A support environment for testing, evaluation, and analysis. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 153–162, Boston, MA, November 1988. ACM SIGSOFT. Appears in *ACM SIGSOFT Notes*, Vol. 13, No. 5.

[5] L.A. Clarke, J.C. Wileden, and A.L. Wolf. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings of the Second International Conference on Ada Applications and Environments*, pages 81–90, Miami Beach, FL, April 1986. IEEE Computer Society.

[6] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[7] A. Evans, K.J. Butler, G. Goos, and W.A. Wulf. *DIANA Reference Manual, Revision 3*. Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[8] K. Forester, I. Shy, and S. Zeil. PARIS operators: An Arcadia perspective. Technical Report Arcadia Document UCI-88-01, Department of Information and Computer Science, University of California at Irvine, 1988.

[9] J.E. Grass and Y. Chen. The C++ information abstractor. In *Proceedings of the Second C++ Conference*, San Francisco, CA, April 1990. USENIX.

[10] A.R. Koenig and B. Stroustrup. Exception handling for C++ (revised). In *Proceedings of the Second C++ Conference*, San Francisco, CA, April 1990. USENIX.

[11] W. Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, Boston, MA, 1975.

[12] J.E.B. Moss and A.L. Wolf. Toward principles of inheritance and subtyping in programming languages. (available as AT&T Bell Laboratories Technical Memorandum 59113-881010-12TM), October 1988.

[13] D.S. Rosenblum. App: An annotation preprocessor for creating self-checking C programs. (in preparation).

[14] D.S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, 2(2):24–33, March 1985.

[15] S. Sankar, D.S. Rosenblum, and R.B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference*, pages 285–296, Paris, France, May 1985. Cambridge University Press.

[16] B. Stroustrup. Parameterized types for C++. In *Proceedings of the C++ Conference*, pages 1–18, Denver, CO, October 1988. USENIX.

[17] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, MA, November 1988. ACM SIGSOFT. Appears in *ACM SIGSOFT Notes*, Vol. 13, No. 5.

[18] J.C. Wileden, L.A. Clarke, and A.L. Wolf. A comparative evaluation of object definition techniques for large prototype systems. *ACM Transactions on Programming Languages and Systems*, 12(4):670–699, October 1990.

[19] A.L. Wolf. Abstraction mechanisms and persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, September 1990.

[20] A.L. Wolf, L.A. Clarke, and J.C. Wileden. The AdaPIC Tool Set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, SE-15(3):250–263, March 1989.

[21] M. Young, R.N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 200–209, Key West, FL, December 1989. ACM SIGSOFT.

[22] S.J. Zeil and E.C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the 10th International Conference on Software Engineering*, pages 241–248, Singapore, April 1988. IEEE Computer Society.