

Arithmetic Equality Constraints as C++ Statements

CHRISTOPHER J. VAN WYK

Department of Mathematics and Computer Science, Drew University, Madison, New Jersey 07940, U.S.A.
and

AT&T Bell Laboratories, Murray Hill, New Jersey 07974, U.S.A.

SUMMARY

Simultaneous equations arise naturally in problems from a variety of application areas. This paper describes how to use two object-oriented features of C++—operator overloading and virtual functions—to allow a programmer to express and to solve a system of simultaneous equations directly in a C++ program.

KEY WORDS C++ Object-oriented programming Operator overloading Simultaneous equations

INTRODUCTION

A computer program in a conventional (imperative) language provides an explicit sequence of instructions by which to compute the desired answer. A program in a declarative language, by contrast, expresses conditions that the answer should satisfy; it is up to an implementation of the language to compute an appropriate answer. The usefulness of declarative programming has been recognized for decades, especially for graphics; Leler¹ summarizes earlier work, including that of Sutherland,² Borning,³ and Van Wyk.⁴ This article describes how one can add some declarative features to the (imperative) C++ language, *using only features of and functions written in the language itself*

A first example

To see the appeal of the declarative approach to programming, consider the following problem:

It costs a widget manufacturer \$1.00 to make a widget that sells for \$1.35. If the manufacturer's fixed overhead is \$300, how many widgets should be sold to earn a profit of \$50?

To solve this problem using an imperative programming language, we would first derive an expression that relates profits to the number of widgets sold: $profits = 0.35 \times numsold - 300$, then solve it for $numsold$ in terms of $profits$:

0038-0644/92/060467-28\$14.00
© 1992 by John Wiley & Sons, Ltd.

Received 13 August 1991
Revised 20 February 1992

$numsold = (profits + 300) \div 0.35$. This last equation is, in effect, an assignment statement that we can use to solve the problem. Were the problem to change, though, perhaps to ask what profits would be if revenues were \$1,000, we would have to derive a different sequence of assignment statements.

This paper describes a C++ class library called `expr.h`, with which we can write the C++ program shown in [Figure 1](#) to express a declarative solution to the problem. The function in [Figure 1](#) first defines four `Expr` objects: `profits`, `revenues`, `costs`, and `numsold`, these represent numbers whose value is to be determined later. Next the function imposes four arithmetic constraints on these four `Expr` objects; the first two equations

```
revenues == numsold * 1.35;
costs == 300 + numsold;           // 300 is fixed overhead
```

are translated from the statement of the problem; the third equation

```
profits + costs == revenues;
```

states a defining relationship among profits, revenues, and costs; and the fourth equation

```
profits == 50;
```

states the profit goal from the problem. These four equations illustrate the key declarative feature offered by class library `expr.h`: arithmetic operators are overloaded so that they can be used to form expressions in `Expr` objects, and the `==` operator is overloaded to assert that the `Expr` objects on its two sides are equal.

Finally, the function in [Figure 1](#) prints the answer, showing two ways to access the values of `Expr` objects. One is to ask an `Expr` object to evaluate and print itself

```
profits.eval () print () ;
```

The other is to ask an `Expr` object to evaluate itself and return its numeric value as a `double`:

```
printf ("%g", numsold.eval () .numval () ) ;
```

```
#include "expr.h"

void business ()
{
    Expr profits, revenues, costs, numsold;
    revenues == numsold * 1.35;
    costs == 300 + numsold;           // 300 is fixed overhead
    profits + costs == revenues;

    profits == 50;

    printf ("To make a profit of $") ;
    profits.eval () print () ;
    printf (", you need to sell") ;
    printf ("%g", numsold.eval () .numval () ) ;
    printf ("\n widgets.\n") ;
}
```

Figure 1. A C++ function whose body includes four simultaneous equations

Executing the function in [Figure 1](#) prints:

```
To make a profit of $50, you need to sell 1000 widgets.
```

To solve the second problem mentioned above, we would merely change the fourth equation in [Figure 1](#) to `revenues == 1000;`.

Overview

The general idea behind class library `expr.h` is to give users a natural notation in which to state equations that relate `Expr` objects. Behind the scenes, the library manipulates these equations using equality-preserving operations like subtracting equal expressions from both sides of an equation. In the best case, these manipulations simplify the equations enough that the values of all `Expr` objects become known. In the worst case, these manipulations simplify none of the equations, and the library can only preserve a record of their having been asserted.

Of course, it is one thing to express a set of constraints, and quite another to find a solution. This paper describes one implementation, which uses a relatively simple algorithm to solve systems of simultaneous equations. Before descending into the details of that implementation, however, the next section shows how to use `expr.h` to write C++ programs in a style strongly reminiscent of IDEAL,⁴ and the following section compares the `expr.h` approach to some other declarative languages. Two sections follow the detailed description of the implementation, the first describing some other implementation possibilities, and the second mentioning some issues related to programming in C++.

IDEAL-STYLE PROGRAMMING

The IDEAL programming language allows line drawings to be described using simultaneous equations that constrain significant points in the drawing.⁴ [Figure 2](#) shows a simple IDEAL program that draws [Figure 3](#). The definition of `rectangle` as a *box* has three parts:

- the definition of six complex variables, four to define the comers of the `rectangle` and two to define its dimensions;
- three equations that constrain the values of these six variables; and
- a *conn* statement that connects the four comers by line segments.

The definition of `box f` begins with a `put`-statement that defines an instance of `rectangle` named `l`. The braces of the `put`-statement reopen the scope of the definition of `rectangle` to impose three more constraint equations on the values of the variables of `rectangle l`.

The definition of `square` as another `box` says that to draw a `square` one draws a `rectangle` named `S` and imposes the additional constraint that the `width` and the `height` of `S` are equal. The second `put`-statement in `f` imposes two constraints on the variables of `square r`, the first of which uses the value of variable `se` in `rectangle l`, and both of which refer to variables of the `rectangle S` that is put by `square`.

[Figure 4](#) shows a C++ program that uses the `expr.h` library to draw the picture in [Figure 3](#). It relies on two functions that affect what we might call ‘the current picture’. Function `connect (Expr, Expr)` adds to the current picture a line segment connecting the two points (complex numbers) that are its arguments; `drawlines ()` displays the contents

of the current picture. The `expr.h` library offers the key advantage that the values of a line segment's endpoints need not be known until `drawlines()` is called.

Figures 2 and 4 exhibit striking parallels and two differences. In both programs, an instance of `rectangle` has six variables constrained by three equations, and adds four line segments to the current picture. One difference between the two programs is cosmetic: the three additional equations imposed on instance `l` of `rectangle` appear inside the instantiation of `l` in the IDEAL program, but outside it in the C++ program. This difference arises

```

box rectangle {
    var nw, ne, sw, se;
    var width, height;
    nw + width = ne;
    sw + width = se;
    sw + (0, 1) * height = nw;
    corm nw to ne to se to sw to nw;
}

box square {
    put s : rectangle {
        width = height;
    }
}

box f {
    put l : rectangle {
        sw = 0;
        width = 1;
        height = 2;
    }
    put r : square {
        S.sw = l.se + (1,1);
        S.width = 1;
    }
}

box main {
    put f{}
}

```

Figure 2. An example IDEAL program

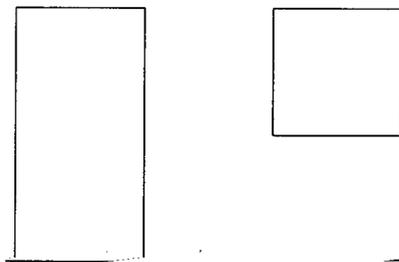


Figure 3. The picture drawn by the IDEAL program in Figure 2

because the C++ program cannot reopen the scope of the class definition; it implies no restriction on the kinds of equations that the user can impose, however, since all members of `rectangle` are public.

The second difference between [Figures 2 and 4](#) is more substantial: the `square` in the IDEAL program *contains* a `rectangle`, while the `square` in the C++ program *is* a `rectangle`. This difference shows up in the additional equations imposed on `square r`, which in the IDEAL program require an extra level of indirection through `S`. Of the two definitions of `square`, the one in the C++ program seems the more natural. Unfortunately

```

#define Complex (x, y)      Expr (x, y)

class rectangle {
public:
    Expr nw, ne, sw, se;
    Expr width, height;
    rectangle ();
    ~rectangle ()      { }
};

rectangle :: rectangle ( )
{
    nw + width == ne;
    sw + width == se;
    sw + Complex (0, 1) *height == nw;
    connect (nw, ne) ;
    connect (ne, se) ;           // Wouldn' t it be nice
    connect (se, sw) ;           // if connect () were
    connect (sw, nw) ;           // a variadic function?
}

class square : public rectangle {
public:
    square ()      { width == height; }
    ~square ()      { }
};

void f ( )
{
    rectangle l;
    l.SW == 0.0;
    l.width == 1;
    l.height == 2;
    square r;
    r.sw == l.se + Complex (1, 1);
    r.width == 1;
}

main ()
{
    f();
    drawlines () ;
}

```

Figure 4. A C++ program to draw Figure 3

it cannot be imitated in the IDEAL program because IDEAL does not allow boxes to be derived from other boxes.

This example shows how closely one can simulate the boxes and `put` -statements of IDEAL using `expr.h`. Other IDEAL features, such as pens for drawing sequences of boxes, opaqueing to clip away parts of the picture, and constructing pictures in multiple overlays, can be implemented in a traditional imperative style. Pens, for example, could be implemented using `for` -loops, while multiple overlays correspond to the ‘canvases’ available in some raster graphics packages.⁵ Thus one can have all the power of C++ available at the same time one has simultaneous equations doing a job they do well.

RELATED WORK

Computer programs for symbolic algebra, such as Macsyma, Maple, and Mathematical, allow one to write systems of simultaneous equations and call routines to solve them, and they provide interfaces to conventional programming languages. The `expr.h` library offers no competition for the powerful solution techniques that come with these packages. On the other hand, the library offers a simple and convenient way to express systems of equations directly in C++ that is unmatched by these systems’ interfaces.

The Bertrand¹ programming language uses augmented term rewriting as a principal control mechanism, and a similar equation solver has been implemented in it.

A program in a logic programming language consists of a set of predicates that are to be satisfied, and it is the job of the language’s implementation to find satisfying assignments to the variables in the predicates. The original Prolog⁶ used unification to seek satisfying variable assignments. Since it distinguished arithmetic predicates from others by the restriction that their operands could not be unknown variables, however, pure Prolog did not offer much help for solving arithmetic equations. More recently, constraint logic programming languages⁷ have added other resolution mechanisms. The implementation of Prolog III,⁸ for example, includes a module that implements the simplex algorithm.

IMPLEMENTATION

The ten subsections of this section explain the implementation of `expr.h`:

- *Useful definitions*: macro and type definitions used throughout the program;
- `Expr`: *the public part*: what a client of the solver needs to know;
- *An equation-solving algorithm*: a short example that illustrates some other features of `Expr` than those shown in [Figures 1](#) and [4](#), then an explanation of this implementation’s equation-solving algorithm;
- `Node` and `Null`: base class for expression-tree data structure;
- `Expr`: *private constructors*: details of the library’s automatic storage management;
- *Constants and variables*: derived classes for leaves of expression trees;
- *Binary arithmetic operators*: some derived classes for internal nodes of expression trees;
- *Ordered linear combinations*: derived classes for the equation-solver’s data structures;
- *Functions*: more derived classes for internal nodes of expression trees;
- *Equation-solving program*: a class that implements the algorithm described in the third section.

The program is written in a highly recursive style. Almost every function does the least work possible, often merely calling another function to do its job. This makes it hard to find a suitable place at which to begin describing it. The bottom-up description here seems to require less ‘reading on faith’ than a top-down approach would.

The description of the implementation also confronts the terminological overlap between simultaneous equations and programming languages. The word ‘variable’, for example, has different but related meanings for the two subject areas. In this section we use ‘variable’ to refer to the abstract idea of a variable in a system of simultaneous equations, and ‘object’ to refer to pieces of storage reserved by a C++ program. In general, words in typewriter-like font are about the program.

Useful definitions

The code in the Figures was extracted directly from the source text of working programs. The code shows how to add constraints to C++, which is only one of several object-oriented languages that allow operator overloading. It would be interesting to compare the code to an implementation in another such language, since each language is a different point in the design space of object-oriented languages.

Figure 5 shows some macro and type definitions that are used throughout the program. The implementation also overloads functions `abs()` and `iszero()` to operate on parameters of type `Expr`. The overloaded definition of `abs()` appears in Figure 17. The overloaded definition of `iszero()` is not shown.

`Expr`: the public part

Figure 6 shows the public part of the definition of class `Expr`. The various constructors create, in order, anonymous and named complex variables, real and complex constants, and complex variables whose real and imaginary parts are the given `Expr` objects. Using the public functions of `Expr`, one can direct an `Expr` object to print or to evaluate itself, and to test whether it is known or numeric; if `numeric()` is true, then `numval()` returns the

```
#define ERROR      fprintf (stderr,
#define WARNING )
#define FATAL     ) , fflush (stdout) , fflush (stderr) , abort ()

#define demand (cond, msg)      if (! (cond)) ERROR msg FATAL

typedef int boolean;

typedef double number;

const number EPSILON = 0.000001;

number abs (number x) { return x >= 0 ? x : -x; }

boolean iszero (number x) { return abs (x) < EPSILON; }
```

Figure 5. Miscellaneous definitions

numeric value of the `Expr` object. This is almost all that a client of class `Expr` needs to know to begin using it.

Class `Expr` serves as a ‘master class’ through which the client can use the equation solver. The arithmetic operators are overloaded to accept `Expr` objects as operands and produce `Expr` objects as results. [Figure 7](#) shows another example of a program that uses `expr.h`. Line 1 implicitly calls the constructor `Expr: :Expr (char *)` to initialize `pi` as a complex variable whose print-name is `pi`. Line 2 implicitly calls the constructor `Expr: :Expr (number)` to promote `3.14159265` from `double` to `Expr`; once both sides are `Expr` objects, the `==` operator implicitly calls the equation solver on its two operands. Line 3 implicitly calls the constructor `Expr: :Expr ()` twice to initialize `diameter` and `circumference` as complex variables. Line 4 implicitly calls a built-in constructor to promote `6` from `int` to `double`, then calls `Expr: :Expr (number)` to promote it to `Expr`; then `==` calls the equation solver as before. Line 5 relies on the overloading of `*` to return an `Expr` object when its operands are `Expr` objects. The last line of

```
class Expr {
public:
    Expr (); // to make ComplexVars
    Expr (char *) ; // to make ComplexVars
    Expr (number) ; // to make Reals
    Expr (number, number) ; // to make Complexes
    Expr (Expr, Expr) ; // to make ComplexVars
    Expr (const Expr&) ; // copy constructor
    void operator= (const Expr &) ;
    ~Expr();
    void print () { p->print () ; }
    Expr eval () { return p->eval () ; }
    boolean known () { return p->known () ; }
    boolean numeric () { return p->numeric () ; }
    number numval () { return p->numval(); }
private:
    Node *p;
    friend class Node;
    // . . .
```

Figure 6. The beginning of the definition of class `Expr`

```
void circle_circumference()
{
    Expr pi = "pi"; // line 1
    pi == 3.14159265; // line 2
    Expr diameter, circumference; // line 3
    circumference == 6; // line 4
    circumference == pi * diameter; // line 5
    printf("A circle with circumference %g",
           circumference.eval() .numval());
    printf(" has radius %g.\n",
           (0.5*diameter) .eval() .numval ());
}
```

Figure 7. More general uses of `Expr` objects

the function illustrates that any `Expr` object—even the product `0.5* diameter`—can use any of the member functions of class `Expr`.

The brief fragment of the private part of the definition of class `Expr`, shown in Figure 6, shows that a variable of class `Expr` contains a pointer `p` to a `Node` object; in fact this is *all* the data that an `Expr` object contains. Figure 6 also shows that an `Expr` object’s public member functions merely delegate their responsibilities to the like-named member function of `p`; the private member functions of `Expr` all do the same. Thus, when reading the code for the equation solver, it helps sometimes to think of an `Expr` object as a ‘pointer to `Node`’.

The solver represents all expressions as trees of `Node` objects. Base class `Node` and the hierarchy of classes derived from it are described in subsequent subsections; meanwhile Table I suggests the meaning of class names that appear in the Figures. None of the constructors or member functions of any of the classes named in Table I is public: the client’s only access to them is through an `Expr` object. The ostensible programming-language justification for this is that clients can neither read nor write any of the data structures used by the solver. In fact, several other advantages also accrue to this arrangement.

First, it is convenient for clients not to need to know the various types of `Node` objects from which expressions are built. The function in Figure 7, for example, uses at least four different types of `Node` objects: `ComplexVar`, `RealVar`, `Real`, and `Product`. The client, however, can think of it merely as using expressions and equations on them.

Second, a client can write a function like

```
void ohms_law (Expr voltage, Expr current, Expr resistance)
{
    voltage == current * resistance;
}
```

and use it anywhere to impose Ohm’s law on any three expressions:

```
Expr V, I, R;
ohms_law (V, I, R) ;
```

Obviously `ohms_law ()` is meant to affect its parameters, and it can because through its `Expr` parameters it receives pointers to the roots of three expression trees. The client, however, need not be concerned about whether the parameters passed to functions should be `Expr` objects, pointers to `Expr` objects, or references to `Expr` objects.

Table I. Overview of the class hierarchy for `Node`

Class Name	Description
<code>Node</code>	base class
<code>Real</code> , <code>Complex</code> , <code>RealVar</code> , <code>ComplexVar</code>	real and complex constants and variables
<code>Sum</code> , <code>Diff</code> , <code>Product</code> , <code>Quotient</code>	binary arithmetic operators
<code>Null</code> , <code>Term</code> , <code>LinComb</code>	parts of ordered linear combinations
<code>RealPart</code> , <code>ImagPart</code> , <code>Abs</code> , <code>Arg</code> , <code>Cosre</code> , <code>Sinre</code> , <code>reSqrt</code> , <code>Cis</code> , <code>Sqrt</code>	functions

Finally, all `Node` objects contain a reference count `refct` that is maintained by the `Expr` member functions shown in Figure 8. Code that uses `Expr` objects instead of pointers to `Node` objects automatically reclaims storage as soon as possible.¹⁰ (Indeed, if one creates one's own pointers to `Node` objects, such references will not be counted; this is almost bound to lead eventually to a catastrophic error.)

Automated memory management is a boon to both the client and the program author, but it does introduce an unfortunate pitfall. The overloading of the assignment operator makes it legal to write '`a + b = c + d;`'. Despite its misleading appearance, this statement does not cause the sum of `a` and `b` to be equated to the sum of `c` and `d`. Instead, it decrements the reference count of the sum of `a` and `b` (which causes that sum to be collected as garbage) and it increments the reference count of the sum of `c` and `d` (so it will never be collected as garbage). In general, we do not assign values to `Expr` objects explicitly; instead we use the equation solver to set their values.

Minimizing type details is another convenience for the client. It does, however, add some complications to the job of the program author. These will become apparent in the descriptions of `Node` and the classes derived from it.

An equation-solving algorithm

The implementation uses the 'slightly non-linear equation solver',¹¹ variations of which have been used in METAFONT,¹² IDEAL,⁴ a picture beautifier,¹³ and the HEQS package for business analysts.¹⁴ The equation-solving algorithm is not the principal focus of the paper, but aspects of it are interwoven throughout the code. Hence, the following brief description will help to explain both the data structures and how `expr.h` solves equations.

Every real variable in the system has a positive integer *serial number* and a *dependency representation*. The dependency representation of a variable is an *ordered linear combination*. This data structure represents a sum of *terms*; a term is the product of a real coefficient and a real variable, or just a real constant (the 'constant term'). The terms in an ordered linear combination are stored in descending order by serial number, with the constant term at the end. The workhorse function of the solver is `add(c1, χ_1 , c2, χ_2)`, where c_i are real constants and χ_i are ordered linear combinations for $i \in \{1, 2\}$; the result of `add(c1, χ_1 , c2, χ_2)` is the ordered linear combination equal to $c_1 \times \chi_1 + c_2 \times \chi_2$. The ordering of terms in the data structure means that `add(c1, χ_1 , c2, χ_2)` can be computed in time proportional to the sum of the number of terms in χ_1 and χ_2 .

```
Expr : ~Expr (const Expr& r) : p (r. p)      { ++p->refct; }

void Expr : operator= (const Expr& r)
{
    ++r. p->refct;
    if (--p->refct == 0)
        delete p;
    p = r.p;
}

Expr : ~Expr ( )      { if (--p-> refct == 0) delete p; }
```

Figure 8. Reference-counting code in `Expr` member functions

All of the solver's variables start out *independent* their ordered linear combinations include only themselves (the initial dependency representation of x is the term $1 \times x$). As equations are solved, variables become *dependent*: their dependency representation contains a linear combination of other variables. Once a variable's dependency representation contains no variables, the variable is *known*: its dependency representation is a constant.

To process a real equation, the solver substitutes the most up-to-date possible dependency representation for each of the variables in the equation. If the resulting equation is linear, the solver chooses one of the variables in it to become dependent on the other variables. If the resulting equation is not linear, the equation is placed on a queue of nonlinear equations. When the system of simultaneous equations is linear, this algorithm amounts to a version of Gauss-Jordan elimination. Since the present implementation processes linear equations as they are seen, linear systems like those in Figures 1 and 4 are solved automatically.

If the system of simultaneous equations is not linear, but a sequence of substitutions can bring each equation into linear form, then the algorithm will still discover the answer. Nonlinear equations are stored on queue `NonLinears`, which is analyzed only when an explicit request that it `solve ()` itself is issued, as shown in the example in Figure 9. In that program, `A [i]`, `B [i]`, and `C [i]` represent the lengths of the sides of triangle i . The equations in the first part of Figure 9 assert that triangles 0, 1, and 2 are similar. The second part of Figure 9 gives the lengths of some of the sides, then directs that any enqueued nonlinear equations should be processed again to see if they become linear after known information has been substituted into them. The last part of Figure 9 prints the following table:

```
void similar_triangles ()
{
    Expr A[3], B[3], C[3]; // lengths of corresponding sides
    for (int i = 0; i < 3; i++) {
        A[i]/A[(i+1) %3] == B[i]/B[(i+1) %3];
        B[i]/B[(i+1) %3] == C[i]/C[(i+1) %3];
        C[i]/C[(i+1) %3] == A[i]/A[(i+1) %3];
    }

    A[0] == 3;
    B[0] == 4;
    C[1] == 10;
    B[2] == 12;
    C[2] == 15;
    NonLinears. solve () ;

    printf ("A\tB\tC\n" ) ;
    for (i = 0; i < 3; i++) {
        A[i].eval().print();
        printf ("\t" ) ;
        B[i].eval().print();
        printf ("\t" ) ;
        C[i].eval().print();
        printf ("\n" ) ;
    }
}
```

Figure 9. A program that includes non-linear equations

A	B	C
3	4	5
6	8	10
9	12	15

For [Figure 9](#), the implementation reduces the equations to linear form not only by substituting in known values, but also by cross-multiplying equations on quotients to form equations on products (that is, $a/b = c/d$ becomes $a \times d = b \times c$).

The present implementation allows variables to take on complex values, which we denote as ordered pairs of real numbers: (a, b) means $a + bi$, where a and b are real and $i = \sqrt{-1}$. The basic strategy is to reduce a complex equation to two real equations—one on the real parts, the other on the imaginary parts—as soon as possible. Thus, a complex variable is created as an ordered pair of real variables. Sums, differences, and products of complex numbers are readily expressed in terms of real numbers:

$$(a, b) \pm (c, d) \text{ becomes } (a \pm c, b \pm d); \quad (1)$$

$$(a, b) \times (c, d) \text{ becomes } (ac - bd, ad + bc). \quad (2)$$

The reduction from complex to real is subtler for quotients and functions, as described below.

Node **and** Null

[Figure 10](#) shows the definition of class `Node` and its member functions. Function `isolc()` returns true only when the node is part of an ordered linear combination. Functions `serial()`, `null()`, `varget()`, `varset()`, `coef f()`, `dependent()`, and `rest()` are used by the equation solver and make sense only when `isolc()` would return true. Functions `isreal()`, `realpart()`, and `imagpart()` are for manipulating complex expressions, while `isquotient()`, `numerator()`, and `denominator()` are used to detect and act on opportunities to cross-multiply the sides of an equation. Functions `name get()` and `name et()` are used to assign and refer to the names of constants and variables. The member functions of `Expr` that correspond to these `Node` functions are all private to `Expr`, so they can be used only by friends of `Expr`.

[Figure 10](#) contains a host of functions beyond the five functions one would expect from [Figure 6](#). Many of these functions pertain to only a few types of `Node` objects. For example, only a `RealVar` object has a serial number; only a `Quotient` object has a numerator and a denominator. The need to provide these member functions in the base class reflects one disadvantage of working with `Expr` objects instead of creating pointers to `Node` objects directly: there is no way to specify that member `p` of an `Expr` object points to a particular type of object derived from class `Node`. Thus, if *any* class derived from `Node` defines a member function, then *all* classes derived from `Node` must define such a function to which an `Expr` object can transfer responsibility. Sometimes we can use this necessity to advantage.

So far as possible, the member functions of base class `Node` are defined to behave innocuously when called on an inappropriate `Node` object. Since real variables have positive serial numbers, `Node::serial()` returns a value that could not possibly be the serial number of a real variable. A `Node` remains silent when asked to print itself. It denies belonging to an ordered linear combination, or being null, dependent, real, known, or numeric; it also denies being a quotient, but if asked nevertheless to decompose itself into

numerator and denominator, it returns itself as its numerator and one as its denominator. Functions `realpart()` and `imagpart()` return unevaluated expression trees. Functions `varget()`, `varset()`, `coeff()`, `nameget()`, `nameset()`, and `numval()` abort execution because it is a serious programming error for them to be called on an inappropriate kind of `Node`.

Evaluating a `Node` object should always return a `Node` object of equal value. Following the Hippocratic principle, ‘First, do no harm’, an arbitrary `Node` object returns itself as the result of `eval()`. Obviously if all `Node` objects relied on the base class definition of `eval()`, not much work would get done. Fortunately, most classes derived from `Node`

```
class Node {
    friend class Expr;
    int refct;
protected:
    Node ( )      : refct (1)                { }
    virtual ~Node ( )                { }
    virtual int serial ( )            { return -1; }
    virtual boolean null ( )          { return 0; }
    virtual Expr varget ( ) ;
    virtual void varset (Expr) ;
    virtual boolean isolc ( )         { return 0; }
    virtual boolean dependent ( )    { return 0; }
    virtual Expr rest ( ) ;
    virtual boolean isreal ( )       { return 0; }
    virtual Expr realpart ( ) ;
    virtual Expr imagpart ( ) ;
    virtual boolean isquotient ( )   { return 0; }
    virtual Expr numerator ( ) ;
    virtual Expr denominator ( ) ;
    virtual char *nameget ( )
        { ERROR "??: : nameget ( ) \n" FATAL; }
    virtual void nameset (char *)
        { ERROR "??: :nameset ( ) \n" FATAL; }
    virtual void print ( )            { }
    virtual Expr eval ( ) ;
    virtual boolean known ( )        { return 0; }
    virtual boolean numeric ( )      { return 0; }
    virtual number numval ( )
        { ERROR "??: : numval()\n" FATAL; }
    virtual number coeff ( )
        { ERROR "??: : coeff()\n" FATAL; }
};

Expr Node::varget ( )                { ERROR "??: :varget()\n" FATAL; }
void Node::varset (Expr)             { ERROR "??: : varset()\n" FATAL; }
Expr Node::rest ( )                  { return NullExpr(); }
Expr Node::realpart ( )              { return Re(this); }
Expr Node::imagpart ( )              { return Im(this); }
Expr Node::numerator ( )             { return this; }
Expr Node::denominator ( )          { return 1.0; }
Expr Node::eval ( )                  { return this; } // primum non nocere
```

Figure 10. Definition of base class `Node`

define their own version of `eval()`, and most define their own versions of several other functions as well: the structure of the solver relies heavily on C++'s virtual functions.

The only member function of `Node` not yet described is `rest()`. Functions in the solver call `rest()` to obtain an expression that represents all but the first term of an ordered linear combination. It would be most convenient if the base-class version of `rest()` returned a 'null expression'. But an `Expr` object whose member `p` is null will attempt to dereference the null pointer when we call any of its member functions. We need instead to provide a distinguished 'null node', which is returned by function `NullExpr()`.

Figure 11 shows the definition of class `Null` and function `NullExpr()`. The member functions of `Null` are defined to work in ways that prove handy for the solver. Thus, the `Null` object is part of an ordered linear combination, and is null, real, known, and numeric, with numeric value zero. The `Null` object reports a serial number of zero so that it can appear as the last term in an ordered linear combination.

Expr: private constructors

The `Expr`-valued function `Node::denominator()` returns `1.0`. This works because the public function `Expr::Expr(number)` is called automatically to transform the number `1.0` into a proper `Expr` object. But what about the `Expr`-valued functions `numerator()` and `eval()`? They return `this`, which is a pointer to a `Node` object.

```
class Null : public Node {
    friend Expr NullExpr ( ) ;
    Null () { }
    'Null () { }
    int serial () { return 0; }
    boolean null () { return 1; }
    Expr target () { return NullExpr ( ) ; }
    boolean isolc () { return 1; }
    boolean isreal () { return 1; }
    Expr realpart () { return 0.0; }
    Expr imagpart () { return 0.0; }
    void print () { printf ("0"); }
    boolean known () { return 1; }
    boolean numeric () { return 1; }
    number numval () { return 0.0; }
    number coeff () { return 0.0; }
};

Expr NullExpr ()
{
    // This is the only instance of a Null in the program.
    // If anybody else-makes one (which shouldn't be possible
    // since this is the only friend of class Null) , then
    // functions that rely on the existence of a single Null
    // will fail in strange ways.
    static Node *TheNull = new Null() ;
    return TheNull;
}
```

Figure 11. Definitions to provide a 'null expression'

Another consequence of using `Expr` objects instead of pointers to `Node` objects is the need for the private constructors shown in [Figure 12](#). These constructors transform a pointer to a `Node` object into an `Expr` object. The first constructor, `Expr::Expr(Node*)`, increments the reference count of its parameter, which points to a `Node` object that already exists, as in `Node::numerator()` and `Node::eval()` in [Figure 10](#).

The second constructor, `Expr::Expr(Node*, freshflag)`, does *not* increment the reference count of its parameter, which should point to a `Node` object that has just been created by `new`; `freshflag` is a dummy class created solely to distinguish the two situations. Were the second constructor not available, and all pointers to `Node` objects handled by the first, then the reference counts of newly created `Node` objects would be doubly incremented, and some storage would never be reclaimed. On the other hand, if the second constructor is called with a pointer to a `Node` object that has not been freshly created, then the failure to maintain the correct reference count will cause premature storage reclamation and subsequent disaster.

These constructors are private members of class `Expr`, so only friends of `Expr` can use them. They are meant to be used only by the solver code, which was written with a careful eye to which of the first two constructors should be called. Clients see the existence of these constructors in one small way: it is safer to write `0.0` than `0` for zero. If one relies on the default constructors to convert `0` to an `Expr` object, the compiler may complain of an attempt to call the private member function `Expr::Expr(Node*)`.

Constants and variables

[Table II](#) depicts the three classes derived from `Node` to store real constants and complex constants and variables. (A description of the class that stores real variables follows immediately.) The part of [Table II](#) immediately below the column headings shows what other data an instance of each class contains. [Table II](#) also shows which member functions each class redefines. The table includes redefinitions that are short enough to fit; a blank means that the class that heads the column does not redefine the function that heads the row. None of these derived classes redefines `null()`, `varget()`, `varset()`, `dependent()`, `rest()`, `isquotient()`, `numerator()` or `denominator()` from their definitions in base class `Node`.

Member functions `nameget()` and `nameset()` manipulate data member `name`, which is used by `print()`. Their implementations are similar to the version for `RealVar` shown in [Figure 14](#).

[Table II](#) shows `Complex::eval()`; [Figure 13](#) shows `ComplexVar::eval()`. To further the goal of reducing from complex to real as soon as possible, when the imaginary parts of an object are known to be zero, both of these evaluation functions return `Expr` objects that point to real expressions. [Figure 13](#) also shows the code for constructors for these three classes, which are called by the various public constructors for class `Expr`.

```
class freshflag { };
Expr::Expr (Node *p) : p (p)           { ++p->refct; }
Expr::Expr (Node *p, freshflag) : p (p) { }
```

Figure 12. Private constructors for class Expr

Table II. Definitions related to constants and complex variables

	Real	Complex	ComplexVar
data members:	char *name; number val;	char *name; number re, im;	char *name; Expr re, im;
serial()	return 0;		
isolc()	return 1;		
isreal()	return 1;	return iszero(im) ;	return iszero(im) ;
realpart()	return val;	return re;	return re;
imagpart()	return 0.0;	return im;	return im;
nameget ()			
nameset()	// See the text for descriptions of these three functions.		
print()			
eval()		if (isreal()) return re; else return this;	// See Figure 13.
known()	return 1;	return 1;	return re.known() && im.known();
numeric()	return 1;		
numval()	return val;		
coeff()	return val;		

```

Expr ComplexVar::eval()
{
    Expr repart = re.eval()
    Expr impart = im.eval()
    if (iszero(impart))
        return repart;
    return Expr(repart, impart) ;
}

// called by Expr::Expr (number)
Real::Real(number x) : val(x), name(0) { }

// called by Expr::Expr (number, number)
Complex::Complex(number x, number y) : re(x), ire(y), name(0) { }

static freshflag FRESH;

// called by Expr::Expr() and Expr::Expr(char *)
ComplexVar::ComplexVar() : re(Expr(new RealVar, FRESH)),
    im(Expr(new RealVar, FRESH)), name(0) { }

// called by Expr::Expr(Expr, Expr)
ComplexVar::ComplexVar (Expr r, Expr i) : re(r), ire(i), name(0) { }

```

Figure 13. ComplexVar::eval() and some constructors

The definitions of `serial()`, `isrc()`, and `coeff()` for class `Real` allow a `Real` object to appear in an ordered linear combination, where it will appear last. Since complex

```

class RealVar: public Node {
    friend class Expr;
    friend solve result operator== (Expr, Expr) ;
    friend class ComplexVar;
    static smax; // max serial number assigned
    int serno; // serial number
    Expr deprep; // dependency representation
    char *name;
    RealVar();
    ~RealVar() { if (name) delete name; }
    int serial() { return serno; }
    Expr varget() { return this; }
    void varset(Expr e) { deprep = e; }
    boolean isrc() { return 1; }
    boolean dependent
        { return deprep.serial() != serial(); }
    boolean isreal() { return 1; }
    Expr realpart() { return deprep; }
    Expr imagpart() { return 0.0; }
    char *nameget() { return name; }
    void nameset(char *s) { name = strdup(s); }
    void print();
    Expr eval();
    boolean known() { return deprep.known(); }
    boolean numeric() { return known(); }
    number numval()
        { if (known()) return deprep.coeff(); }
    number coeff() { return deprep.coeff(); }
};

Expr maketerm(number c, Expr v)
    { return Expr(new Term(c, v), FRESH); }

RealVar::RealVar ()
    : serno(++smax), deprep(maketerm(1.0, this)), name(0) { }

void RealVar::print()
{
    if (name)
        printf("%s", nameget());
    else
        printf("UNK#%d" serial());
}

Expr RealVar::eval()
{
    if (dependent) // substitute latest info available
        deprep = deprep.eval();
    return deprep;
}

```

Figure 14. Definitions related to real variables

objects cannot appear in ordered linear combinations, classes `Complex` and `ComplexVar` inherit `serial()`, `isolc()`, and `coeff()` from base class `Node`.

Figure 14 shows code related to class `RealVar`, where real variables are stored. The data members of a `RealVar` object include `serno`, the real variable's serial number, and `deprep`, the dependency representation of the value of the variable. When a `RealVar` object is created, it is assigned a serial number and its dependency representation is initialized to one times itself; thus, the new variable is independent. When the variable no longer appears in its own dependency representation, it is dependent, as computed by `dependent()`. The variable finally is known when its dependency representation is known, as computed by `known()`. The solver uses functions `varget()` and `varset()` to obtain a pointer to the variable and to change its dependency representation.

A `RealVar` object prints itself either by printing its name or, if it is anonymous, by printing its serial number (UNK stands for 'unknown'). A `RealVar` object evaluates itself by first evaluating its dependency representation; this recursive computation removes all dependent variables from any dependency representations that it examines, so `eval()` always returns an ordered linear combination of independent variables. Such a 'lazy' approach to maintaining dependency representations is considerably simpler than the eager approach used in earlier implementations of the solver,¹¹ which required that variables be eliminated from all dependency representations in the system as soon as they became dependent. It was inspired by a shift from viewing `deprep` as a data structure to understanding it as an object that could evaluate itself.¹⁵

Binary arithmetic operators

Figure 15 shows the definition of class `Sum` and the overloading of the `+` operator. Notice first that all `+` does when one of its operands is an `Expr` object is to return an expression tree whose root is a `Sum` object, and whose children are the operands of the `+`. Evaluation of a `Sum` object proceeds as follows. If the operands of the `Sum` object are not both real, then its value is computed as two real sums—one on the real parts and one on the imaginary parts. If both operands are real, there are three possibilities:

- both operands are ordered linear combinations: `add()` computes their sum as the value of the `Sum` object;
- one of the operands is zero: the value of the `Sum` object is the other operand, even if that operand is not an ordered linear combination;
- neither of these two cases applies: the value of the `Sum` object is a `Sum` object whose operands are the results of evaluating the operands of the original `Sum` object (which the calls to `eval()` may have simplified somewhat from their original form).

(Notice, incidentally, how little some of the member functions of `Sum` do. Functions `realpart()` and `imagpart()`, for example, leave their operands to do almost all the work. Even worse, function `isreal()` is a lazy pessimist: it returns true only in the easy case when both operands of the `Sum` are known to be real. All of these functions are private to `Expr` to avoid clients' being misled by their answers.)

The code for `Diff`, `Product`, and `Quotient` is similar to that for `Sum`. Function `Product::eval()` checks for possible simplifications when an operand is zero or one. Function `Quotient::eval()` checks for division by zero, as well as possible simplifications when the numerator is zero or the denominator is one. The `realpart()`

and `imagpart()` functions for `Sum`, `Diff`, and `Product` implement the well-known Rules (1) and (2). If we always apply the general rule for complex division

$$\frac{(a,b)}{(c,d)} \text{ becomes } \frac{(ac + bd, bc - ad)}{(c^2 + d^2)}, \quad (3)$$

however, then every evaluation of a quotient of unknowns doubles the degree of the denominator. Whenever the divisor is real, `Quotient::realpart()` and `Quotient::imagpart()` apply instead the simpler rule

$$\frac{(a,b)}{c} \text{ becomes } \left[\frac{a}{c}, \frac{b}{c} \right] \quad (4)$$

```
class Sum: public Node {
    friend class Expr;
    friend class LinComb;
    friend Expr operator+(Expr, Expr) ;
    Expr left, right;
    Sum (Expr, Expr) ;
    ~Sumo;
    boolean isreal()
        { return left.isreal() && right.isreal(); }
    Expr realpart()
        { return left.realpart() + right.realwrt(); }
    Expr imagpart()
        { return left.imagpart() + right.imagpart(); }
    void print() ;
    Expr eval() ;
};

Sum: : Sum (Expr l, Expr r)          : left (l), right(r)      {}

Expr Sum: :evalo
{
    if (isreal()) {
        Expr l = left.eval() ;
        Expr r = right.eval() ;
        if (l.isolc() && r.isolc())
            return add(1.0, l, 1.0, r);
        if (l.numeric() && iszero(l.numval ()))
            return r;
        if (r.numeric() && iszero(r.numval ()))
            return l;
        return l + r;
    }
    return Expr(realpart() , imagpart()) .eval() ;
}

Expr operator+(Expr l, Expr r)
    { return Expr(new Sum(l, r), FRESH); }
```

Figure 15. Definitions related to addition

This apparently minor touch in fact leads to substantial savings, because after one application of Rule (3) the denominator of a quotient *is* real.

Class `Quotient` also redefines `isquotient()`, `numerator()`, and `denominator()` in the obvious way.

Ordered linear combinations

Table III and Figure 16 describe the implementation of classes `LinComb` and `Term`, the data structures that store ordered linear combinations. A `Term` object is the product of a real coefficient and a real variable; the coefficient appears as the left operand of the product, and the variable as the right operand; the `Term` object represents a real constant if the variable is null. Functions `Term::coeff()` and `Term::varget()` return the term's coefficient and variable, respectively; `Term::serial()` is the same as its variable's serial number.

A `LinComb` object stores an ordered linear combination as a sum; its left operand is the first `Term` object of the ordered linear combination, and its right operand is an ordered linear combination of the remaining `Term` objects. Functions `LinComb::varget()` and `LinComb::coeff()` return the variable and the coefficient in the first term of the ordered linear combination; function `LinComb::rest()` returns the ordered linear combination less its first term. The ordering property of ordered linear combinations means that if `lc` is a `LinComb` object, then `lc.serial() > lc.rest().serial()`.

Table III. Classes for storing ordered linear combinations

	LinComb	Term
derived from:	sum	Product
left. p should point to a:	Term	Real
right. p should point to a:	LinComb, Real, or Null	RealVar or Null
<code>serial()</code>	return left. serial () ;	return right. serial () ;
<code>varget()</code>	return left. varget () ;	return right .varget () ;
<code>isolc()</code>	return 1;	return 1;
<code>rest()</code>	return right;	
<code>isreal()</code>	return 1;	return 1;
<code>realpart()</code>	return this;	return this;
<code>imagpart()</code>	return 0.0;	return 0.0;
<code>eval()</code>	// See Figure 16.	
<code>known()</code>	return left. known () && right null () ;	return right. null () ;
<code>numeric()</code>	return left numeric () ;	return right. numeric () ;
<code>coeff()</code>	return left. coeff () ;	return left. coeff () ;

The constant term of an ordered linear combination can be stored as a `Null` object (when it is zero), as a `Real` object, or as a `Term` object whose variable is null. This flexibility makes it easier to write functions that manipulate the data structure.

Figure 16 also shows the code for function `add()`, which forms a linear combination of two ordered linear combinations, `x1` and `x2`. The expression returned by `add()` is an ordered linear combination: its `Term` objects appear in decreasing order by serial number; `add()` also eliminates terms with zero coefficients from the result.

Functions

The solver provides the nine single-parameter functions shown in Table IV. It uses the same general strategy to evaluate all of them. If the function's parameter is known, evaluation returns the function's value on that parameter; otherwise it returns an unevaluated

```

Expr LinComb: : eval ()
{
    return add (1 .0, left. eval(), 1.0, right.eval());
}

Expr Term: : eval ()
{
    return add(coeff(), varget().eval(), 1.0, NullExpr());
}

Expr makelincomb (number c, Expr v, Expr n)
{
    return Expr (new LinComb (maketerm (c, v) , n) , FRESH) ;
}

Expr add (number c1, Expr x1, number c2, Expr x2)
{
    demand (x1.isolc () , "x1 is not in canonical linear form" ) ;
    demand (x2.isolc () , "x2 is not in canonical linear form" ) ;
    if (x1.serial() == 0 && x2.serial() == 0) // constant term
        return c1*x1.coeff () + c2*x2.coeff () ;
    number coeff; // leading coefficient
    if (x1.serial() == x2.serial()) {
        coeff = c1*x1.coeff () + c2*x2.coeff () ;
        if (!iszero(coeff))
            return makelincomb(coeff, x1.varget () ,
                add(c1, x1.rest(), c2, x2.rest()));
        else return add(c1, x1.rest(), c2, x2.rest());
    } else if (x1.serial() > x2.serial()) {
        coeff = c1*x1.coeff();
        if (!iszero(coeff))
            return makelincomb(coeff, x1.varget(),
                add(c1, x1.rest(), c2, x2));
        else return add(c1, x1.rest(), c2, x2);
    }
    else { // x1.serial() < x2.serial()
        coeff = c2*x2.coeff();
        if (!iszero(coeff))
            return makelincomb(coeff, x2.varget(),
                add(c1, x1, c2, x2.rest()));
        else return add(c1, x1, c2, x2.rest());
    }
}

```

Figure 16. Functions related to ordered linear combinations

expression tree. Thus, to add a function to the program, we must define a node suitable for representing the expression tree.

Figure 17 shows the code related to the absolute value function; the code for other functions looks very similar. An `Abs` object has room for a single parameter. Function `abs(z)` simply returns an `Expr` object that points to an `Abs` object whose `param` is `z`.

Table IV. Functions provided by the solver

Function	Description
<code>Re(z)</code>	the real part of (a,b) is a
<code>Im(z)</code>	the imaginary part of (a,b) is b (notice that the imaginary part is real)
<code>abs(z)</code>	the absolute value of (a,b) is the length of the vector from the origin to the point (a,b)
<code>arg(z)</code>	the argument of (a,b) is the angle the vector (a,b) makes with respect to the positive x -axis
<code>cosre(theta)</code>	the cosine of a real number θ
<code>sinre(theta)</code>	the sine of a real number θ
<code>resqrt(x)</code>	the nonnegative square root of a nonnegative real number x
<code>cis(theta)</code>	the unit vector that lies at angle θ with respect to the positive x -axis
<code>sqrt(z)</code>	the square root in the upper half of the complex plane of a complex number z

```

class Abs: public Node {
    friend class Expr;
    friend Expr abs (Expr) ;
    Expr param;
    Abs (Expr p) : param(p) { }
    ~Abs ()          { }
    Expr eval() ;
    boolean isreal() { return 1; }
    Expr realpart() { return this; }
    Expr imagpart() { return 0.0; }
    void print()
        { printf ("abs (" ; param. print () ; printf (") ") ; }
};

Expr Abs::eval()
{
    Expr x = Re (param).eval() ;
    Expr y = Im (param).eval() ;
    if (x. numeric() && y.numeric())
        return sqrt (x .numval() *x. numval() +
                    y.numval()*y. numval());
    return this;
}

Expr abs (Expr z )          { return Expr (new Abs (z) , FRESH) ; }

```

Figure 17. Definitions related to the absolute value function

Since absolute value is a real-valued function it is easy to define `isreal()`, `realpart()`, and `imagpart()`. If `Abs::eval()` succeeds in evaluating both the real and complex parts of `param`, it computes and returns the absolute value of the parameter otherwise it returns the original `Abs` object (as it was before evaluation).

There are two reasons why the program defines special real-valued versions of the cosine, sine, and square-root functions. First, the implementation does not support complex trigonometric functions: both `cosre()` and `sinre()` ignore any complex part of their parameters; the suffix `re` is a reminder of this limitation. Second, and more important, the solver must be able to tell when a function value is real in order to reduce from complex to real whenever possible. Consider the rule for complex square root:

$$\sqrt{(a,b)} \text{ becomes } \sqrt{|(a,b)|} \exp((i/2)\tan^{-1}(b/a)). \quad (5)$$

To avoid an infinite recursion when this rule is applied, the solver must be able to tell that the argument to the square root on the right side of Rule (5) is nonnegative, so that the value of the square root is real. The easiest way to arrange this is to use the special function `resqrt()` to separate it out as a special case.

```
enum solverresult { INCONSISTENT = 0, CONSISTENT = 1, NONLINEAR };

solverresult equatereals (Expr left, Expr right)
{
    demand (left.isreal() ,
            "equatereals () got a non-real left" ) ;
    demand (right.isreal() ,
            "equatereals () got a non-real right" ) ;
    Expr zero = (left - right).eval() ;
    if (zero.numeric() )
        return is zero (zero.numval()) ?
            CONSISTENT : INCONSISTENT ;
    if (!zero.isolc())
        return NONLINEAR;
    Expr elim = findterm (zero) ;
    number coeff = elim.coeff () ;
    Expr var = elim.varget() ;
    demand (! var.dependent() , "can only eliminate indep vars" ) ;
    var.varset (add (-1.0/coeff, zero, 1.0,
                    makelincomb (1.0, var, NullExpr () ) ) ) ;
    return CONSISTENT;
}

EqnQueue Nonlinear;

solverresult operator== (Expr left, Expr right)
{
    solverresult retval = equatecomplexes (left, right);
    if (retval == NONLINEAR)
        NonLinears.enqueue (left, right);
    return retval;
}
```

Figure 18. Code for solving linear equations

Equation-solving program

Figure 18 shows how linear equations are solved. The `==` operator and its auxiliary functions, `equatereals()` and `equatecomplexes()`, return a value from enum `solveresult` that indicates the disposition of the equation:

- `CONSISTENT` equations have been imposed on the variables;
- `INCONSISTENT` equations have been ignored;
- `NONLINEAR` equations have been enqueued to be tried again later.

Notice that all of Figures 1, 4, 5, and 9 ignore the value returned by `==`; this risky practice generates a warning from the compiler. The more careful programmer at least would check that an equation did not return `INCONSISTENT`.

Auxiliary function `equatecomplexes()` (not shown) passes the real and imaginary parts of both its parameters in pairs to `equatereals()` and returns the more pessimistic of the two values returned by `equatereals()`. Function `equatereals()` evaluates the difference between its left and right operands. If this difference is a number, then the equation is either inconsistent or redundant (trivially consistent). Otherwise, one of the variables in the difference is chosen to become dependent on the remaining variables. In emulation of partial pivoting in Gaussian elimination, `findterm()` (not shown) selects a variable whose coefficient is largest in absolute value to become dependent on the others.

Figure 19 shows how the queue of nonlinear equations is implemented. An `eqblk` is a linked-list node with room for two `Expr` operands, `left` and `right`, which are to be set equal. Function `EqnQueue::solve()` traverses the queue of equations repeatedly as follows. First, the distinguished `sentinel` is posted to mark the current end of the queue. Next, each equation in turn is passed to `equatecomplexes()` ;

- if the equation turned out to be linear, then some progress has been made towards solution, and the equation need not be considered again;
- if it was nonlinear but `massage()` rearranged the components of the equation, that counts as progress, and the massaged equation is enqueued to be further simplified on the next trip through the queue;
- otherwise the original equation is enqueued;

in all cases, the equation is removed from the head of the queue. If `sentinel` is ever encountered when `progress` is zero, then further traversal of the queue is pointless: too much nonlinearity remains in the system to be removed by the slightly nonlinear solver.

IMPLEMENTATION ALTERNATIVES

The program described in the previous section shows one way to add arithmetic equations and equation-solving to C++. This section explains briefly some other ways one might try to do the job.

Mutable types

Almost all of the types in the program are immutable: no member of an object changes once the object has been initialized. The sole exception is `RealVar`, whose member `deprep` is

updated as equations are solved. Under this policy, one can pass any `Expr` object to any function without fear that the `Expr` object will be changed.

```

class EqnQueue {
    eqblk *head, *tail;
    static eqblk sentinel;
    void postsentinel() ;
    void unpostsentinel() ;
public:
    EqnQueue ( )          { head = tail = 0; }
    void enqueue(Expr l, Expr r) ;
    void solve() ;
};

boolean message(Expr left, Expr r right,
    Expr *newleft, Expr *newright)
{
    if (left.isquotient() || right.isquotient()) {
        *newleft = left.numerator()
            * right.denominator() ;
        *newright = right.numerator()
            * left.denominator() ;
        return 1;
    }
    return 0;
}

void EqnQueue: :solve()
{
    boolean progress = 1;
    while (progress) {
        progress = 0;
        postsentinel ();
        while (head != &sentinel) {
            Expr nl = NullExpr(), nr = NullExpr();
            if (equatecomplexes (head->left,
                head->right) != NONLINEAR)
                progress = 1;
            else if (message (head->left, head->right,
                &nl, &nr)) {
                NonLinears .enqueue(nl, nr);
                progress = 1;
            } else
                NonLinears. enqueue (head->left,
                    head->right) ;
            eqblk *ohead = head;
            head = head->next;
            delete ohead;
            if (!head)
                tail = 0;
        }
        unpostsentinel() ;
    }
}

```

Figure 19. Code for processing nonlinear equations

Immutability introduces some awkwardness into programming. Almost every use of an `Expr` object outside of the simultaneous equations needs to be modified by `eval()`, `numval()`, or both. If `Expr` objects were mutable, one might be able to write

```
Expr x[10];
// equations in terms of x[i]
double y = x[3];           // instead of x[3].eval().numval()
```

This could conflict with the lazy evaluation strategy, however, since `x [3]` might be known only after a call to `x [3].eval()`. Moreover, it is very demanding to expect the type system both to promote built-in types to user-defined types and to demote them in the other direction whenever possible, all automatically. The variety of possible combinations of types makes implementation a daunting prospect.¹⁶

Another disadvantage of immutability is that common subexpressions are evaluated repeatedly. As a dramatic illustration, consider the program in [Figure 20](#), which prints

```
1
(1 + 1)
((1+1) +1)
```

So long as it guaranteed the arithmetic equality of `Expr` objects, one could certainly adopt a more flexible policy on their mutability, trading off generality against efficiency in symbolic computation.

Two-level hierarchy

An earlier version of this program worked with real variables and equations.¹⁵ An obvious idea would be to build the complex variables and equations on top of that package, perhaps allowing the user to declare both `RExprs` (real expressions) and `CExprs` (complex expressions). This would require developing two parallel hierarchies of classes and operators. For example, there would be class `RSum`, whose evaluation function looked like the inner part of `Sum::eval()`, and `CSum`, whose evaluation function broke complex sums down into two `RSum` objects; the `+` operator would need to be doubly overloaded: `operator+ (RExpr, RExpr)` and `operator+ (CExpr, CExpr)`. While it is tempting to hope that many of the conversions between real and complex would be handled automatically by the C++ type mechanism, some, such as recognizing when the result of a square-root is real, probably would still need to be specified explicitly by the program.

```
void peano_arithmetic ( )
{
    Expr a = 1;
    while (a. eval() .numval ( ) < 4) {
        a.print();
        printf ("\n" );
        a = a + 1 ;
    }
}
```

Figure 20. Old-fashioned counting

Other solving algorithms

One might try to enhance the solver's capabilities by adding more equation rearrangements. For example, one might rewrite $c = \sqrt{x}$ as $c^2 = x$ when c is a constant. Adding such rules to `massage()` must be done carefully given the simplistic control structure used by `EqnQueue::solve()`. When `massage()` succeeds in rearranging the equation, `EqnQueue::solve()` calls that progress, and enqueues the massaged equation *instead of* the original; it enqueues only one equation because if it enqueued both equations but never reduced either to linear then the queue would continue to grow indefinitely. Thus, rearrangement rules added to `massage()` should never leave an equation harder to solve than it was before the rearrangement. An alternative would be to devise a more sophisticated control structure for `EqnQueue::solve()`.

Whatever symbolic methods were added to the solver, it probably would still not solve many systems of nonlinear equations. One could still use the overloaded operators to assert the equations, then some scheme adapted to a particular kind of nonlinear system. Global Newton iteration has served well in another graphics context,¹⁷ and would be convenient since the equations are available in symbolic form. The slightly nonlinear solver could still be useful in wringing as much linearity as possible from the system before passing it on to the nonlinear solver.

NOTES ON C++

Pitfalls

Some of the errors that one can easily make when using `expr.h` arise from the limitations imposed by using C++ to parse the equations. For example, if one writes `x == y == z`, then `z` is equated to a member of the `enum solve result`, which probably is not what one intended. The correct way to write the complex number $(0, 1)$ is `Expr (0, 1)`. The tempting `(0, 1)` just invokes the comma operator and returns a real `1`. Another way to write $(0, 1)$ is `sqrt (Expr (-1))`. The equally tempting `sqrt (-1)` causes the `math.h` function `sqrt()` to be called with a negative argument.

Memory management algorithms that use reference counts do not collect inaccessible structures that are circularly linked. An independent variable, however, *is* a circularly linked structure. This means that the storage is never reclaimed for variables that remain independent after the solver is finished. It also explains the initializations of `n1` and `nr` in `EqnQueue::solve()`.

'Transposing' a class hierarchy

Sometimes it is useful to see the definitions of all the member functions of a class. At other times it is useful to see all the definitions of a virtual member function in the various classes that redefine it. A tool that would form a matrix of member functions and classes like [Tables II](#) or [III](#) would help to make both views of a C++ program equally accessible.

cost

Consider a 10×10 dense linear system, which represents 20 real equations in 20 variables. The solver creates over 100,000 `Exprs` and calls `malloc()` around 8,000 times when it solves such a system. Since the time complexity of Gaussian elimination is cubic in the number of variables, the solver evidently does not impose an undue constant factor on the algorithm.

ACKNOWLEDGMENTS

John Hobby and Brian Kernighan offered attentive ears and helpful suggestions while I was writing `expr.h`. They also commented on drafts of this paper, as did Jon Bentley, Andrew Koenig, Rob Pike, and Howard Trickey.

REFERENCES

1. Wm Leler, *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, Reading, Massachusetts, 1988.
2. Ivan Sutherland, 'SKETCHPAD: a man-machine graphical communication system', *IFIPS Proceedings of the Spring Joint Computer Conference, 1963*.
3. Alan Borning, 'The programming language aspects of ThingLab, a constraint-oriented simulation laboratory', *ACM Transactions on Programming Languages and Systems*, **3**, (4), 353–387, 1981.
4. Christopher J. Van Wyk, 'A high-level language for specifying pictures', *ACM Transactions on Graphics*, **1**, (2), 163–182, 1982.
5. James A. Foley, Andries van Dam, James Feiner, and John Hughes, *Computer Graphics: Principles and Practice*, Second edition, Addison-Wesley, Reading, Massachusetts, 1990.
6. William F. Clocksin and Christopher S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
7. Jacques Cohen, 'Constraint logic programming languages', *Communications of the ACM*, **33**, (7), 52–68, 1990.
8. Alain Colmerauer, 'An introduction to Prolog III', *Communications of the ACM*, **33**, (7), 69–90, 1990.
9. Grady Booth, *Object-Oriented Design with Applications*, Benjamin/Cummings, Reading, Massachusetts, 1991.
10. Andrew R. Koenig, 'An example of dynamic binding in C++', *Journal of Object-Oriented Programming*, **1**, 60–62, 1988.
11. Emanuel Derman and Christopher J. Van Wyk, 'A simple equation solver and its application to financial modeling', *Software—Practice and Experience*, **14**, (12), 1169–1184, 1984.
12. Donald E. Knuth, *METAFONT: The Program*, Volume D of *Computers & Typesetting*, Addison-Wesley, Reading, Massachusetts, 1986.
13. Theo Pavlidis and Christopher J. Van Wyk, 'An automatic beautifier for drawings and illustrations', *Computer Graphics*, **19**, (3), 225–234, 1985.
14. E. Derman and E. G. Sheppard, 'HEQS—a hierarchical equation solver', *AT&T Technical Journal*, **64**, (9), 2061–2096, 1985.
15. Christopher J. Van Wyk, 'A class library for solving simultaneous equations', *Proceedings of the Third USENIX C++ Technical Conference*, 1991, pp. 229–234.
16. Brian W. Kernighan, 'An AWK to C++ Translator', *Proceedings of the Third USENIX C++ Technical Conference*, 1991, pp. 217–228.
17. Greg Nelson, 'Juno, a constraint-based graphics system', *Computer Graphics*, **19**, (3), 235–243, 1985.