A Simple Solver for Linear Equations Containing Nonlinear Operators

Norman Ramsey*
Bell Communications Research
445 South St, Morristown, NJ 07960, USA

November 8, 1996

Abstract

This paper presents a simple equation solver. The solver finds solutions for sets of linear equations extended with several nonlinear operators, including integer division and modulus, sign extension, and bit slicing. The solver uses a new technique called *balancing*, which can eliminate some nonlinear operators from a set of equations before applying Gaussian elimination. The solver's principal advantages are its simplicity and its ability to handle some nonlinear operators, including nonlinear functions of more than one variable.

The solver is part of an application generator that provides encoding and decoding of machine instructions based on equational specifications. The solver is presented not as pseudo code but as a literate program, which guarantees that the code shown in the paper is the same code that is actually used. Using real code exposes more detail than using pseudocode, but literate-programming techniques help manage the detail. The detail should benefit readers who want to implement their own solvers based on the techniques presented here.

INTRODUCTION

Using very-high-level specifications and programming languages can help make programmers more productive, as well as helping them write programs that are more reliable, easier to read, and easier to maintain. For example, where low-level systems make programmers say *how* to establish relationships among a set of variables, high-level systems can use equations or constraints to say *what* relationships should hold (Sutherland 1963; Borning 1981; Knuth 1986b; Ladd and Ramming 1994). This paper describes a general-purpose, arithmetic equation solver. The solver is used in a toolkit that transforms equational

*Author's current address: Dept of Computer Sciences, 1398 Computer Science Building, Purdue University, West Lafayette, IN 47907

specifications of machine instructions into C or Modula-3 programs that encode or decode those instructions (Ramsey and Fernandez 1995). The solver is a vital component of the machine-code toolkit; using equations helps guarantee that the encoding and decoding software are consistent. It would be nearly impossible to provide such a guarantee without using equational specifications.

Many techniques exist for solving different kinds of equational systems. Algebraic equations can be used to define operators; term-rewriting techniques are often used to work with algebraic systems. This paper focuses on arithmetic equations, in which the meanings of all the operators are defined in advance, and solutions are sought in integers or real numbers. When such equations are linear, one can find real solutions by simple Gaussian elimination. Finding integer solutions requires a bit more care since simple division is not always possible.

Other useful software tools have been built around extensions of Gaussian elimination. For example, Metafont, a tool for generating digital letterforms, uses equations to constrain the relative positions of the parts of letters (Knuth 1986b). Its solver uses a variant of Gaussian elimination over the reals, plus an extension that records when two expressions are known to be equal (Knuth 1986a, §585). The HEQS solver extends Knuth's technique by adding a rewriting step that transforms some nonlinear equations into linear ones, e.g., rewriting $1 = 3/a$ to $a = 3$ (Derman and Van Wyk 1984). If the HEQS solver can't immediately rewrite an equation into linear form, it enqueues the equation and retries it later.

This paper presents a different extension to Gaussian elimination, one which handles "balanceable" nonlinear operators. The idea is to move nonlinear operators from the equations into data structures called *balances*, which tell how to solve a simple nonlinear equation using a function and its inverse. If enough nonlinear operators can be removed, the equations can be solved using ordinary linear techniques. For example, if we have the equation $x = y + 2 \times \cos\theta$, we introduce a fresh variable θ_1 to stand for $\cos\theta$, and we note the "balanced" relationship $\theta_1 = \cos\theta$ and $\theta = \arccos\theta_1$. The equation becomes $x = y + 2 \times \theta_1$, which is a linear equation that can be solved with Gaussian elimination, and the balance is used either to compute θ given θ_1, or to compute θ_1 given θ. The balancing technique applies to any invertible function with multiple inputs and multiple outputs.

The solver described in this paper is used to analyze machine-code descriptions that use two balanceable nonlinear operators. The bit-slicing operator, e.g., $n[3{:}7]$, extracts bits 3 through 7 of the two's-complement representation of the integer n. The sign-extension operator, e.g., $n\!\uparrow_{16}$, takes the least significant 16 bits of n as a two's-complement signed integer. Two more balanceable operators, integer division and modulus, are introduced by the solver in the process of restricting solutions to the integers.

Balancing is not the only way to handle bit slicing, sign extension, division, and modulus. The last part of this paper shows how to rewrite "mostly linear" equations involving these operators as integer linear programs. Although integer linear programming is an NP-complete problem (Papadimitriou and Steiglitz 1982), there are powerful, practical methods for solving some classes of integer linear programs (Pugh 1992). These methods are complex; substantial effort is needed to implement them, and sometimes even to reuse existing implementations. In a special-purpose application, little language, program transformer, or application generator, it may be cheaper to implement a simpler technique, like the balancing technique, than to create an interface to a more general implementation that already exists. This choice has worked well in the machine-code toolkit.

This paper shows how to implement Gaussian elimination with balancing. After a description of balancing, the implementation appears in three parts: the basic solving engine, which is like that of Knuth (Knuth 1986a), the extension that uses balances to eliminate terms containing nonlinear operators, and the code that creates balances. The concepts and code are illustrated with example equations and balances taken from machine descriptions.

A Literate Program

This paper not only describes an implementation of a balancing equation solver, it *is* the implementation. The `noweb` system (Ramsey 1994) for literate programming (Knuth 1984) extracts this paper and a working implementation from a single source. This source contains the prose of the article interleaved with named "code chunks." The code chunks are written in the order best suited to describing the solver, not the order dictated by a compiler. Chunks contain source code and references to other chunks. The names of chunks appear italicized and in angle brackets:

3a ⟨*summarize the problem* 3a⟩≡ 3c ▷

```
writes("Inputs are: ")
⟨for every i that is an input, print i and a blank 3b⟩
write()
```

3b ⟨*for every* `i` *that is an input, print* `i` *and a blank* 3b⟩≡ (3a)

```
every i := !inputs do
  writes(i, " ")
```

The ≡ sign indicates the definition of a chunk. Definitions of a chunk can be continued in a later chunk; `noweb` concatenates their contents. Such a concatenation is indicated by a +≡ sign in the definition:

3c ⟨*summarize the problem* 3a⟩+≡ ◁ 3a

```
write("Starting to solve ", *eqns, " equations")
```

`noweb` adds navigational aids to the paper. Each chunk name ends with the number of the page on which the chunk's definition begins. When more than one definition appears on a page, they are distinguished by appending lower-case letters to the page number. When a chunk's definition is continued, `noweb` includes pointers to the previous and next definitions, written "◁3a" and "3c▷." The notation "(3a)" shows where a chunk is used. Chunks with no numbers are included in the source file, but they don't appear in this paper. For purposes of exposition, such chunks play the role of pseudo-code, but unlike pseudo-code they have real implementations.

The complete source code for the solver, as extracted from this paper, is available for anonymous `ftp` from Internet host `ftp.cs.purdue.edu` in the directory `pub/nr/solver`. This paper includes just enough chunks to show how to implement a balancing equation solver, without overwhelming readers with detail. The advantage of presenting a literate program instead of just an algorithm is that readers see (part of) a working implementation; because the source code in the paper is compiled and produces a working solver, they can be confident that the paper is correct. The level of detail should enable other implementors of little languages and application generators to build solvers based on the program presented here; I have written the paper that I wished I could read when I was writing my own solver.

Icon

The solver shown in this paper is written in the Icon programming language (Griswold and Griswold 1990). Icon has several advantages for writing an equation solver: its high-level data structures (lists, sets, and tables) provide direct implementations of mathematically sophisticated ideas, its automatic memory management simplifies algebraic manipulation, and its *goal-directed evaluation* of expressions makes it easy to write searching code, e.g., to find a variable to eliminate. Such searching code relies on *generators*, which may generate multiple values; for example, the expression `!S` generates all the elements of the set (or list) `S`. Predicates of the form "there exists an `x` in `S` such that `P(x)`" may be written idiomatically as `x := !S & P(x)`. Goal-directed evaluation causes the generator `!S` to be re-evaluated until it produces an `x` satisfying `P(x)`. If no such `x` exists, the expression *fails*. In expressions with multiple generators, Icon automatically backtracks until it finds a combination of generated values satisfying all predicates.

The other Icon idiom with which readers may not be familiar is used to perform some action on every element of a set or list S, which one does by writing

```
every x := !S do ⟨action on x⟩
```

Other details of Icon that are needed to understand the solver are explained as they are encountered.

EQUATIONS WITH NONLINEAR OPERATORS

To illustrate the techniques involved in balancing nonlinear operators, I have taken equations and operators from the description of the MIPS R3000 architecture in the New Jersey Machine-Code Toolkit (Ramsey and Fernandez 1994). The nonlinear operators used to describe machine code include widening (sign extension), narrowing, bit slicing, and integer division and modulus. The first three operate on a two's-complement representation of integers. Widening, or $n\!\uparrow_k$, takes the low-order k bits of a two's-complement integer n and considers those k bits themselves as a two's-complement integer. This operation, also called sign extension, is commonly applied to some fields of machine instructions to produce signed operands. Narrowing, or $n\!\downarrow_k$, is the inverse operation, considering only the low-order k bits of the integer n. The narrowing operation succeeds only if it can be performed without loss of information. Bit slicing, or $n[i{:}j]$, considers bits i through j of n as an unsigned integer, where bit 0 is the least significant bit. The difference between $n[0{:}k-1]$ and $n\!\downarrow_k$ is that $n[0{:}k-1]$ always succeeds, whereas $n\!\downarrow_k$ contains an implicit assertion that the value of n fits in k bits, and the narrow fails if that assertion does not hold. This section shows how these operators are used in equations that describe machine instructions, and it shows what it means to "balance" the operators.

Sign extension is used in the description of the MIPS load instruction, which loads a word into register rt:

$$\text{load } rt, \mathit{offset}, \mathit{base}.$$

The operands rt, offset and base are used to compute the corresponding fields of the instruction, which we write in **bold** font (**rt**, **offset** and **base**). The distinction between operands and fields is what differentiates the assembly-language and machine-language forms of an instruction. The New Jersey Machine-Code Toolkit uses the solver described here to transform a set of operands to a set of fields, or vice versa. The transformation is specified by the following equations, which indicate that the operand offset is obtained by sign-extending the 16-bit field **offset**:

$$\begin{aligned} \mathbf{rt} &= rt \\ \mathbf{offset}\!\uparrow_{16} &= \mathit{offset} \\ \mathbf{base} &= \mathit{base} \end{aligned}$$

These equations can be used by themselves as the basis for a fields-to-operands (decoding) transformation. For the inverse transformation, we solve for the fields in terms of the operands:

$$\begin{aligned} \mathbf{rt} &= rt \\ \mathbf{offset} &= \mathit{offset}\!\downarrow_{16} \\ \mathbf{base} &= \mathit{base} \end{aligned}$$

The result shows we compute the value of the **offset** field by narrowing the offset operand to 16 bits.

Bit slicing is used to describe the MIPS jump instruction, whose sole operand is the target address target. The architecture manual specifies that this target address is computed by taking its most significant four bits from the current program counter, using zero as its least significant two bits, and taking the remaining bits from the 26-bit **target** field:

$$\begin{aligned} \mathit{target}[28{:}31] &= \mathtt{pc}[28{:}31] \\ \mathit{target}[0{:}1] &= 0 \\ \mathit{target}[2{:}27] &= \mathbf{target} \end{aligned}$$

When encoding, we know `pc` and target, so the third equation provides a value for the field **target**, while the first two become constraints on the value of the operand target. When decoding, however, we know `pc` and **target**, and we need all three equations to come up with this solution for target:

$$\mathit{target} = 2^{28} \times \mathtt{pc}[28{:}31] + 2^2 \times \mathbf{target}.$$

Integer division and modulus don't appear explicitly in the descriptions of MIPS instructions, but they have to be introduced to solve the equations describing relative branches. Like jumps, branches have a single *target* operand, but the target address is computed by sign-extending the **offset** field, multiplying it by 4, and adding it to the address of the instruction in the delay slot, which is $\texttt{pc} + 4$:

$$target = (\texttt{pc} + 4) + 4 \times \mathbf{offset}\!\uparrow_{16}.$$

Solving this equation for encoding yields a way to compute the **offset** field, plus a constraint on the operand *target*:

$$\begin{array}{rrcl} & \mathbf{offset} & = & ((target - \texttt{pc} - 4) \text{ div } 4)\!\downarrow_{16} \\ \text{Constrain} & (target - \texttt{pc} - 4) \bmod 4 & = & 0 \end{array}$$

In these examples, one of two sets of variables (operands or fields) is known initially, and we have to solve for the others. In each case, we can "invert" or "balance" a term involving a nonlinear operator so that we are always using the nonlinear operators only to compute functions of known variables. By choosing the proper half of the balance, we can solve for nonlinear terms in a linear solver. We express the balance relationship by introducing fresh variables for nonlinear terms, and writing *balancing equations* for those variables. So, for example, when we see $\mathbf{offset}\!\uparrow_{16} = \mathit{offset}$, we introduce the fresh variable $\mathbf{offset}_1$ to stand for $\mathbf{offset}\!\uparrow_{16}$, and we write $\mathbf{offset}_1 \bowtie \mathbf{offset}$ (pronounced "offset-sub-1 balances offset"). More completely, we write equations by which the variables on either side of the balance can be computed using only the values of the variables on the opposite side:

$$\mathbf{offset}_1 = \mathbf{offset}\!\uparrow_{16} \bowtie \mathbf{offset} = \mathbf{offset}_1\!\downarrow_{16}$$

We write a balance for the slicing example this way, introducing fresh variables $target_1$, $target_2$, and $target_3$:

$$\begin{array}{c} target_1 = target[0{:}1], target_2 = target[2{:}27], target_3 = target[28{:}31] \\ \bowtie \\ target = target_1 + 2^{28} \times target_3 + 2^2 \times target_2 \end{array}$$

Again, when the variables on either side of the $\bowtie$ are known, the variables on the other side can be computed.

Finally, the balance for division by 4 has this form:

$$q = n \text{ div } 4, r = n \bmod 4 \bowtie n = 4 \times q + r$$

The solver described in this paper works in two steps. The first step introduces fresh variables to stand for nonlinear terms, creating balances describing those variables. The second step finds a solution given the equations and the balances, using a Knuth-style eliminator with an extension to use the balances. The first step depends on the set of nonlinear operators chosen, but the second step is general. Because of that generality, and because it is easier to understand the technique by learning how balances are used before learning where they come from, I discuss the eliminator first.

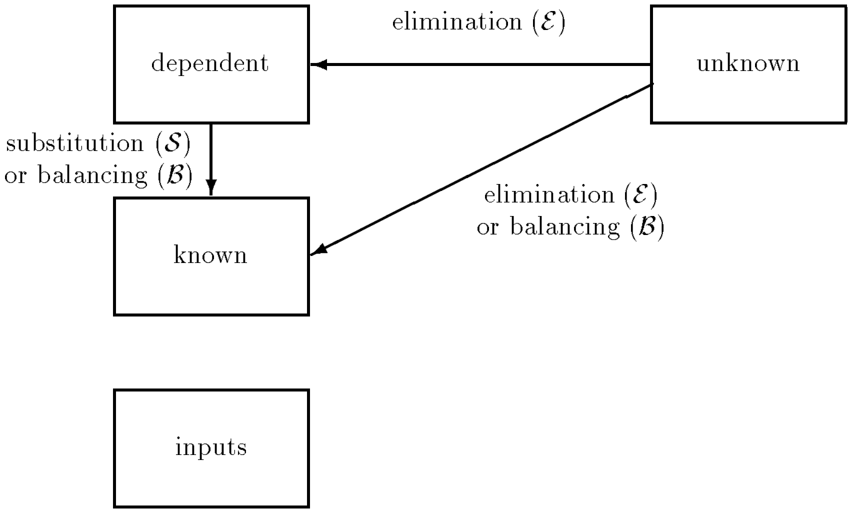

Figure 1: Different states of variables

SOLVING BALANCED EQUATIONS

This section describes the basic solving engine. Both the terminology for describing the technique and the general outline of the implementation come from Knuth's implementation of Gaussian elimination (1986a), §585. There is a modest difference in that Knuth's solver provides numerical answers directly, whereas my solver uses a level of indirection. Certain variables of the equations are designated as *inputs*, and the solver finds expressions that compute the values of other variables as functions of the inputs. From these expressions one can generate C or Modula-3 code to solve any instance of the problem described by the equations. In the degenerate case in which no variables are designated as inputs, my solver behaves as does Knuth's solver, finding a numerical solution.

Figure 1 shows how the variables of the equations make transitions among the states used by the solver. All variables other than the designated inputs are initially *unknown*, which means nothing is known about their values. As the solver works, they may become dependent or known. A *dependent* variable is a function of other variables, either inputs or unknowns. A *known* variable is function of inputs alone. The sets on the left side of Figure 1 overlap; every input is known, and every known variable is either dependent or an input.

An unknown variable becomes dependent by Gaussian elimination of that variable; if the expression giving the value of that variable contains only inputs, then that variable is known. Dependent variables become known when all the unknowns on which they depend have been eliminated; For example, if $y = x - z$, and z is found (by elimination) to be $x - 4$, then when $x - 4$ is substituted for z, y is now known to be 4. When all the variables on one side of a balance become known, the balance is "activated," i.e., the solver uses the balance's equations to make the variables on the other side known, too.

We use two other special terms. Should there be more equations than variables, and should the extra equations not be redundant, some equations become *constraints* on the inputs. Finally, we call an expression *computable* if it is a function of inputs. This definition of "computable" is not standard.

Manipulating expressions by computer is complicated when the same expression can be represented in many different ways. In a typical tree representation, an expression like $2b + 3c + 4 + 5a$ can have over a hundred different equivalent representations, which vary only in the order in which operands are added or multiplied (Derman and Van Wyk 1984). Icon enables a simpler representation. The solver represents variables as strings and integers as integers. It represents sums of linear terms as Icon tables, or associative arrays, in which keys are variables (or expressions) and values are their coefficients. A constant term is represented using the special key 1. For example, we create a table `t` representing $2 \times x + 4$ by

```
t      := table(0)
t["x"] := 2
t[1]   := 4
```

The `0` argument to `table()` is a default value; using this default makes it possible to look up the coefficient of any variable not appearing in `t` and get the answer `0`.

The names of variables make good keys because Icon strings are atomic values; two strings with the same characters are the same string. Keys in Icon tables are not ordered, so this representation of linear sums is a canonical form. This representation also makes it easy to take linear combinations of linear sums and to substitute a linear sum for a variable. Nonlinear operators are represented as Icon records with capitalized names; for example, $n\uparrow_{16}$ is `Widen("n", 16)`.

Our choice of representation ameliorates but does not eliminate the problem of equivalent representations that do not compare equal. Distinct tables never compare equal, even when they have the same keys and values, and distinct records never compare equal, even when they are of the same type and have the same field values. As a result, equality tests require a hand-written procedure, not just Icon's built-in equality operator.

The solver maintains the following data structures:

`inputs`	The set of input variables.
`value`	A table in which the keys are known variables and the values are expressions giving the values of those variables in terms of inputs and unknowns. `value` initially contains only inputs; the value of an input `x` is `x`.
`zeroes`	A list of expressions that are known to be zero. It is computed initially by subtracting the right-hand sides of the equations from the left-hand sides.
`pending`	A temporary list, holding zero-valued expressions from which no variable can be eliminated. These `pending` expressions are returned to `zeroes` at the end of each step of the solving algorithm.
`constraints`	A list of constraints that must be satisfied if the equations are to have a solution. They come from equations in which all variables are known. The list is initially empty.

The solver's job is to transfer information from `zeroes` to `value` and `constraints`. It looks at the zero-valued expressions in `zeroes` one step at a time. At each step, it tries to eliminate a variable, while maintaining the following invariants:

I. No variable in `zeroes`, `pending`, or `constraints` is dependent.

II. No variable appearing in an expression in `value` is dependent.

III. If all the variables on one side of a balance are known, then all the variables on the other side of that balance are known.

IV. Values in `zeroes`, `pending`, `constraints`, and `value` are all represented as sums of linear terms.

Invariant III is maintained by the balance machinery, which is described in the next section. The solver communicates with the balance machinery by adding variables to the queue `newlyknown` as they become known.

Invariant IV is a detail of the implementation; because sums are represented as Icon tables, they are passed by reference to the substitution procedure, which can modify them. If a variable x were represented simply as x rather than $1 \times x$, it would be represented by a string, passed by value, and the substitution procedure would not be able to modify it.

Here is the outline of the solver, which shows the initialization of its data structures:

11 ⟨*solver.icn** 11⟩≡ 14c ▷

```
procedure solve(eqns, balances, inputs)
  value  := table()     # values of dependent variables
  constraints := []     # constraints to check
  zeroes      := []     # expressions equal to zero
  pending     := []     # pending equations we couldn't solve earlier

  every x := !inputs do
    value[x] := term2sum(x)
  every eq := !eqns do
    put(zeroes, subtract(eq.left, eq.right)) # append difference to 'zeroes'

  ⟨initialize balance machinery ⟩
  ⟨if inputs make one side of balances known, make the other sides known 17⟩
  ⟨take expressions from zeroes, eliminating a variable at each step (⋆) 12a⟩
  ⟨make sure all dependent variables are known ⟩
  ⟨build and return solution ⟩
end
```

The Icon notation `[]` denotes an empty list. `value[x]` means "the slot in table `value` indexed by `x`;" the assignments to `value[x]` initialize the `value` table to hold the inputs. Using `term2sum` maintains invariant IV. A hash mark "`#`" introduces a comment, which continues to the end of the line.

The chunk marked (⋆) is the main solving step; preceding chunks are preliminary, and succeeding chunks check to see that the equations have indeed been solved. Unnumbered chunks, like ⟨*initialize balance machinery* ⟩, don't appear in this paper.

The solver makes progress by looking for unknown variables to eliminate. (If there are no unknown variables, then all dependent variables must be known, and we have a solution.) For guidance, chunks are marked ($\mathcal{E}$), ($\mathcal{S}$), and ($\mathcal{B}$) when they correspond to the elimination, substitution, and balancing steps depicted in Figure 1.

To find a variable to eliminate, the solver looks at each expression `z` in `zeroes`. Because this solver is designed to find integer solutions, a variable `v` can be eliminated only if it has a unit coefficient (± 1). If no variable can be eliminated, but the expression `z` does have an unknown variable, it goes into `pending`, because future substitution for a dependent variable may make it possible to eliminate the unknown. (A floating-point solver would use a different strategy, e.g., choosing to eliminate the variable with the largest coefficient. There would be no need for `pending`, and the details of the elimination step would differ from those shown here, but the rest of the solver would be the same.) Finally, if `z` doesn't have any unknown variables, it is a constraint.

12a ⟨*take expressions from* `zeroes`*, eliminating a variable at each step* ($\star$) 12a⟩≡ (11)

```
while ⟨zeroes is not empty⟩ do {
  while z := get(zeroes) do {
    ⟨normalize z by dividing out the greatest common divisor of all coefficients⟩
    if ⟨there is a v in z that can be eliminated (ℰ)⟩ then {
      ⟨make progress by eliminating variable v from z (ℰ) 12b⟩
    } else if ⟨z has an unknown variable⟩ then {
      put(pending, z)
    } else {
      put(constraints, zero_to_constraint(z))
    }
  }
  ⟨if a pending expression can be fixed with div or mod, drain pending 23a⟩
}
⟨if there are pending expressions, fail (with a suitable error message)⟩
```

The main elimination step ($\mathcal{E}$) takes place in three stages. First, if a variable `v` in `z` can be eliminated, it becomes dependent, and we can compute a value for it. To transform `z` into the value of `v`, either add `v` to `z`, or subtract `v` from `z` and negate the result, depending on the sign of `v`'s coefficient.

12b ⟨*make progress by eliminating variable* `v` *from* `z` ($\mathcal{E}$) 12b⟩≡ (12a) 12c▹

```
⟨transform z into the value of v⟩
value[v] := z
```

After installing `v` in the `value` table, we substitute the new value for every occurrence of `v`, maintaining invariant I. Invariant II is maintained automatically, by virtue of invariant I. Because the substitution might enable an elimination that had been impossible, we move all the `pending` expressions back to `zeroes`.

12c ⟨*make progress by eliminating variable* `v` *from* `z` ($\mathcal{E}$) 12b⟩+≡ (12a) ◃12b 13▹

```
⟨substitute z for v in zeroes, value, pending, and constraints (S)⟩
⟨return pending expressions to zeroes⟩
```

Finally, if `v` is not just dependent but known, we put it in `newlyknown` and activate the balance mechanism. The balance mechanism checks to see if the newly known variable *completes* the balance, that is, if it makes all the variables on one side of the balance known.

13 ⟨*make progress by eliminating variable* `v` *from* `z` ($\mathcal{E}$) 12b⟩+≡ (12a) ◃12c

```
if computable(inputs, value[v]) then
  put(newlyknown, v)
⟨while a variable in newlyknown completes a balance, use it to make more variables known (B) 14a⟩
```

When the solver finishes, the `value` table contains the values of all the variables; the solver returns these values, plus any constraints that may have been generated.

We can illustrate the operation of the solver with a simple example containing two variables and no inputs:

$$\begin{array}{rcrcr} 2 \times x & + & 3 \times y & = & 11 \\ x & - & y & = & -2 \\ x & + & 2 \times y & = & 7 \end{array}$$

After subtracting the right- from the left-hand sides, the solver reaches a solution in four steps:

1. The solver considers $2 \times x + 3 \times y - 11 = 0$, cannot find a variable to eliminate, and adds it to `pending`.
2. x can be eliminated from $x - y + 2 = 0$, yielding $x = y - 2$, which the solver adds to `value`. It also substitutes $y - 2$ for x in the pending and remaining equations, and it appends the pending equation to `zeroes`, which now represents:

$$\begin{array}{rcrcr} 3 \times y & - & 9 & = & 0 \\ 5 \times y & - & 15 & = & 0 \end{array}$$

3. Normalizing $3 \times y - 9 = 0$ and eliminating y yields $y = 3$. With substitution, `value` now holds $x = 1$ and $y = 3$.
4. Substitution has transformed the last equation to $0 = 0$, which has no unknown variables and therefore becomes a constraint. A post-solver pass over the solution eliminates such trivially satisfied constraints.

This example shows all three of the alternatives in the main loop ($\star$): elimination, transfer to `pending`, and transformation into a constraint.

USING BALANCES TO MAKE VARIABLES KNOWN

We use a naïve algorithm to determine when a variable completes a balance: whenever a variable first becomes known, check all balances in which that variable appears to see if there is a side of the balance on which all variables are known. Because of the variety of ways in which a variable can become known, it is easiest to take a conservative approach to identifying newly known variables. When the solver makes a variable known by elimination, it adds it to the queue `newlyknown`. Substitution or balancing might make more variables known. For efficiency's sake, we only check balances against variables that have just become known. We identify such variables by checking to see if they are in the set `alreadyknown`, into which we insert each variable as it becomes known. Both `newlyknown` and `alreadyknown` are initially empty.

14a ⟨*while a variable in* `newlyknown` *completes a balance, use it to make more variables known* ($\mathcal{B}$) 14a⟩≡ (13 17)

```
while v := get(newlyknown) do
  if not member(alreadyknown, v) then {
    insert(alreadyknown, v)
    ⟨if v completes half of a balance, make the other half known 14b⟩
    ⟨add every known variable in value to newlyknown ⟩
  }
```

The final step, ⟨*add every known variable in* `value` *to* `newlyknown` ⟩, requires a search through the entire `value` table. The cost of this step could be reduced by adding extra data structures and marking changed variables in step ($\mathcal{S}$).

The task remaining is twofold: discover if a variable completes a balance, and if so, make the variables on the other side known. We could simply check each balance to see if `v` completes it, but when all the variables on one side of a balance are known, that completes the balance, which makes all the variables on the other side known, which also completes the balance, which could lead to infinite regress. We avoid this outcome by deleting each balance from `balances` when it is completed. No information is lost because all the variables in a complete balance are known and therefore appear in `value`.

14b ⟨*if* `v` *completes half of a balance, make the other half known* 14b⟩≡ (14a)

```
every b := ⟨a balance in which v appears ⟩ do
  if complete := can_complete_balance(b, value, inputs) then {
    ⟨remove b from balances and other sets in which it may appear ⟩
    ⟨take the unknown variables from complete balance complete and make them known 16a⟩
}
```

To see if `v` completes a balance, we need to know how balances are represented. The left- and right-hand sides of a `balance` are lists of `balitems`, where a `balitem` holds a variable and an expression for the value of that variable.

14c ⟨*solver.icn** 11⟩+≡ ◁11 15▷

```
record balance(left, right)      # lists of balitem
record balitem(v, value)         # v is string, value is exp
```

For example, we represent the balance

$$\mathbf{offset}_1 = \mathbf{offset}\!\uparrow_{16} \bowtie \mathbf{offset} = \mathbf{offset}_1\!\downarrow_{16}$$

as

```
balance( [balitem("offset#1", Widen ("offset",   16))],
         [balitem("offset",   Narrow("offset#1", 16))] ).
```

The only variables in the `value` fields are the variables listed on the opposite side of the balance. The solver assumes that variable names used in balances don't collide with other names; whatever agent provides the balances must make it so.

The procedure `can_complete_balance` is a predicate that succeeds only if balance `bal` is actually complete. (Instead of returning truth values, Icon predicates work by succeeding—in which case they return a result—or by failing.) If `bal` is complete, `complete_balance` puts the side with all known variables on the left:

15 ⟨*solver.icn** 11⟩+≡ ◁14c

```
record complete_balance(known, unknown) # lists of balitem

procedure can_complete_balance(bal, value, inputs)
  local vl, vr
  if ⟨there is a variable vl on the left of bal that is not computable⟩ then
    if ⟨there is a variable vr on the right of bal that is not computable⟩ then
      fail # balance is not complete
    else
      return complete_balance(bal.right, bal.left)
  else
    return complete_balance(bal.left, bal.right)
end
```

The sense of the tests may seem unusual. It is idiomatic Icon; Icon provides a natural and efficient way to write predicates of the form "there exists an x such that $P(x)$", but no analogous way to write predicates of the form "for all x, $P(x)$." The procedure `can_complete_balance` should return a `complete_balance` if there is a side on which all the variables are computable. By negating the test, reversing the `then` and `else` branches of the `if` statement, and applying de Morgan's laws, we can ask instead if there exists a variable that is not computable, a question we can ask efficiently in Icon.

Once a complete balance is identified, each unknown variable `u` can be made known using the `value` field of the balance item. The `value` field expresses `u` as a function of the known variables on the other side of the balance. Because these variables are known, their values are in the `value` table, and we can get the value of `u` by substitution. `subst_tab` does the substitution, and `term2sum` preserves invariant IV. Because `u` has become known, we add it to `newlyknown`.

16a ⟨*take the unknown variables from complete balance* `complete` *and make them known* 16a⟩≡ (14b)

```
every u_item := !complete.unknown do {
  u := u_item.v            # the variable to be made known
  u_val := term2sum(subst_tab(u_item.value, value))
  ⟨make u's value u_val, without losing current information about u's value 16b⟩
  put(newlyknown, u)
}
```

The chunk ⟨*make* `u`*'s value* `u_val`*, without losing current information about* `u`*'s value* 16b⟩ implements the state transitions labelled ($\mathcal{B}$) in Figure 1. We can't just assign `u_val` to `value[u]`, because `value[u]` may already contain vital information about `u`. As Figure 1 shows, `u` may be known, dependent, or unknown, so there are three ways to treat `u_val`. If `u` is known, then the new value must be consistent with the old one, and we have a new constraint. (`u` remains known, so no state transition is shown in Figure 1.)

16b ⟨*make* `u`*'s value* `u_val`*, without losing current information about* `u`*'s value* 16b⟩≡ (16a) 16c▷

```
if ⟨u is already known⟩ then {
  put(constraints, eqn(value[u], "=", u_val)) # new constraint
}
```

If `u` is dependent, the new value still must be consistent with the old, but we have a new equation, not a new constraint.

16c ⟨*make* `u`*'s value* `u_val`*, without losing current information about* `u`*'s value* 16b⟩+≡ (16a) ◁16b 16d▷

```
else if ⟨u is dependent⟩ then {
  put(zeroes, subtract(value[u], u_val))   # new eqn value[u] = u_val
  value[u] := u_val                        # make u a known variable
}
```

Only if `u` is unknown can we simply assign `u_val` to `value[u]`, and then we must remember to substitute `u_val` for all occurrences of `u`.

16d ⟨*make* `u`*'s value* `u_val`*, without losing current information about* `u`*'s value* 16b⟩+≡ (16a) ◁16c

```
else {
  value[u] := u_val                       # make u dependent
  ⟨substitute u_val for u in zeroes, pending, value, and constraints⟩
}
```

The inputs themselves can complete a balance without any variable ever becoming known. The solver handles inputs as it handles newly known variables, by executing this code, which runs after the solver initializes the balance machinery but before it looks at `zeroes`:

17 ⟨*if inputs make one side of balances known, make the other sides known* 17⟩≡ (11)
⟨*put every element of* `inputs` *into* `newlyknown` ⟩
⟨*while a variable in* `newlyknown` *completes a balance, use it to make more variables known* ($\mathcal{B}$) 14a⟩

Balances provide equations for computing the variables on either side given the values of the variables on the other. From this representation, one might think that the solver needs to know the details of how one side is computed from the other, but in fact the balances could equally well act as placeholders that simply record the dependence of one set of variables on the other. One would record such dependence by using "opaque" operators in the balances; such operators could be used to generate code even if their meanings were unknown to the solver and balancer.

Now that we've seen how balances are used in solving, it's time to consider where they come from.

INTRODUCING BALANCES

We introduce balances in order to eliminate nonlinear operators from equations to be solved. We do it in a rewriting step, then pass the rewritten equations and the balances to the solver described above. An algorithm for introducing a balance can be described in three parts: *substitution rules* that show how to replace nonlinear terms with fresh variables, the *balance* that is introduced, and *equations* that relate the fresh variables to existing variables. Before considering implementation, let's look at the rules for "balancing out" narrow and widen, integer division and modulus, and bit slicing.

When we see an expression widened from or narrowed to a constant width, we can substitute a variable for the widen or narrow and introduce a balance. Using mnemonic names, the rules for eliminating a widen operator are:

$$
\begin{aligned}
e{\uparrow}_k &\mapsto w && (1)\\
w = n{\uparrow}_k &\bowtie n = w{\downarrow}_k && (2)\\
n &= e && (3)
\end{aligned}
$$

where w stands for a wide variable, n for a narrow variable, e for an expression, and k for an integer constant. The meanings of the rules are:

(1) When an expression of the form $e{\uparrow}_k$ is found, introduce a fresh variable w to stand for that expression, and replace that expression by w everywhere it occurs.

(2) Introduce another fresh variable n, and add the balance $w \bowtie n$, showing how to compute n from w or w from n.

(3) Add the equation $n = e$.

The rules for eliminating narrow are similar:

$$
\begin{aligned}
e{\downarrow}_k &\mapsto n\\
w = n{\uparrow}_k &\bowtie n = w{\downarrow}_k\\
w &= e
\end{aligned}
$$

The balancer can eliminate division and modulus by an integer constant k, provided both division and modulus appear in the equations.

$$
\begin{aligned}
e \text{ div } k &\mapsto q\\
e \bmod k &\mapsto m\\
d = q \times k + m &\bowtie q = d \text{ div } k, m = d \bmod k\\
d &= e
\end{aligned}
$$

d is the dividend, q is the quotient, and m is the modulus.

The balancer can eliminate bit slices over constant ranges, provided that the slices cover the full range of the value being sliced (e.g., 32 bits). In general, slices can overlap and be nested inside one another, which leads to tricky code. For purposes of this paper, we restrict our attention to the common case in which the ranges $[l_i : h_i]$ partition the bits of the value being sliced. For example, the ranges $\mathit{target}[0{:}1]$, $\mathit{target}[2{:}27]$, and $\mathit{target}[28{:}31]$ partition a 32-bit address. In such a case, we can write:

$$
\begin{array}{rcl}
e[l_1 : h_1] & \mapsto & s_1 \\
 & \vdots & \\
e[l_n : h_n] & \mapsto & s_n \\
w = \sum_i 2^{l_i} \times s_i & \bowtie & s_1 = w[l_1 : h_1], \ldots, s_n = w[l_n : h_n] \\
w & = & e
\end{array}
$$

w is the wide value, each s_i is a slice, and l_i and h_i are the low and high ends of the range over which slice s_i is taken.

The rewriting step

We can introduce a balance if and only if all its substitution rules can be applied. For example, it is pointless to introduce a balance if we see only $e \bmod 4$, but we should introduce a balance if we see both $e \bmod 4$ and $e \operatorname{div} 4$, because the two together are necessary and sufficient to determine e. We use a three-pass process to introduce balances:

A. *Find applicable substitutions.* Find sub-expressions of forms for which we might substitute. Create fresh variables to stand for such sub-expressions, and save the sub-expressions and variables in special tables.

B. *Add balances and equations.* Examine the tables used in step A, and when a balance can be introduced, do so. When introducing a balance, add its associated equations to the list of all equations, and add its associated substitutions to the table `balmap`, which maps expressions to variables of the balance. In general, only some of the substitutions found in step A are added to `balmap`.

C. *Perform substitutions.* Using `balmap`, perform all of the substitutions associated with the balances introduced in step B.

Steps A, B, and C can be considered separately for each kind of balance. We begin with the supporting structure.

The implementation of the balancer in Icon is complicated by Icon's lack of nested procedures, which forces the auxiliary tables used in steps A and B to be passed as arguments to several functions. Aside from these tables, the balancer uses three data structures. `balances` is a set of balances added. `neweqns` is a list of equations; we make a copy of the original equations because the balancer may add to and modify them. As explained above, `balmap` holds substitutions.

20a ⟨*balancer.icn** 20a⟩≡ 20b ▷

```
procedure balance_eqns(eqns)
  ⟨initialize auxiliary tables to empty tables⟩
  ⟨A. apply balpass1 to every sub-expression of eqns, modifying auxiliary tables⟩
  balances := set()
  neweqns := copy(eqns)         # don't modify the original equations
  balmap := table()
  ⟨B. use auxiliary tables to add to balances, neweqns, and balmap 21⟩
  ⟨C. replace every sub-expression of neweqns that appears in balmap with the corresponding variable⟩
  return [neweqns, balances]
end
```

The chunks performing steps A–C are labelled appropriately. Steps A and C use utility functions that walk the expression data structures; step B is implemented completely within the body of `balance_eqns`.

Step A is implemented by applying the function `balpass1` to every sub-expression of every equation. `balpass1` looks at the type of a sub-expression `e`, which indicates whether a substitution rule might apply. The chunk named ⟨*, auxiliary tables*⟩ stands for a list of the auxiliary tables used in steps A and B.

20b ⟨*balancer.icn** 20a⟩+≡ ◁ 20a

```
procedure balpass1(e ⟨, auxiliary tables⟩)
  case type(e) of {
    ⟨cases for types indicating opportunities for substitution 20c⟩
  }
end
```

We can now show all three steps used to introduce balances for widens and narrows. First, we recognize the forms:

20c ⟨*cases for types indicating opportunities for substitution* 20c⟩≡ (20b) 22a ▷

```
"Widen"  : addkset(widens,  var_for(e.n, varaux, varmap), e)
"Narrow" : addkset(narrows, var_for(e.n, varaux, varmap), e)
```

The parser used with the solver guarantees that the widths given with `Widen` and `Narrow` are integer constants, so we don't need to test that property here. `var_for` is the procedure that creates fresh variables. It uses two auxiliary tables, `varaux` and `varmap`. `varaux` keeps enough state to ensure that

$$\texttt{var\_for(e1, varaux, varmap)} \equiv \texttt{var\_for(e2, varaux, varmap)}$$

whenever `e1` and `e2` represent the same expression. `varmap` provides an inverse transformation, so for any `e`, `varmap[var_for(e, varaux, varmap)]` is a set of expressions equivalent to `e`.

`widens` and `narrows` are the auxiliary tables used to remember what widens and narrows have been seen. They use a pair of keys `[v, k]`, where `v` is the fresh variable introduced for the expression widened, and `k` is the number of bits used in the widen. For example, if we see something of the form $e{\uparrow}_k$ and we introduce the fresh variable n to stand for e, then the table lookup `widens[n, k]` produces the set of all sub-expressions equivalent to $e{\uparrow}_k$. There can be many such sub-expressions because every application of the `Widen` record constructor produces a distinct value. The function `addkset` adds elements to a two-level table whose elements are sets.

We can introduce a balance for each widen and each narrow. Widens and narrows are unusual in that the same balance might possibly be introduced both by a widen and by a narrow. Duplicate balances would result in unnecessary constraints in the solver. The constraints would be tautological, but it is easier to recognize and eliminate duplicate balances than to recognize tautologies. To do so, we keep track of balances in the table `bal_narrow_widen`.

The `every` statement uses the `widens` table to find all possible values of n and k such that we can introduce the balance

$$w = n{\uparrow}_k \bowtie n = w{\downarrow}_k.$$

21 ⟨B. *use auxiliary tables to add to* `balances`, `neweqns`, *and* `balmap` 21⟩≡ (20a) 22b ▷

```
bal_narrow_widen := table()
every n := key(widens) & k := key(widens[n]) do {
  ws := widens[n, k]                  # set of all expressions of form Widen(n, k)
  w  := var_for(?ws, varaux, varmap) # fresh var for any such expression
  if ⟨there is no balance n ⋈ w in bal_narrow_widen⟩ then {
    ⟨add new balance n ⋈ w to bal_narrow_widen⟩
    insert(balances, balance([balitem(w, Widen (n, k))],
                             [balitem(n, Narrow(w, k))]))
  }
  add_equation(neweqns, n, varmap[n])
  every balmap[!ws] := w
}
```

The last two lines take care of the equations and substitution rules needed to eliminate widens, as described on page 18. One adds the equation $n = e$, and the other updates `balmap` so that in step C the variable w will be substituted for every expression equivalent to $e{\uparrow}_k$.

The code for adding balances using `narrows` is similar and need not be shown.

The strategy used to find opportunities to eliminate division and modulus is like that used to eliminate narrows and widens. Step A uses auxiliary tables `divs` and `mods`, which are organized on the same principle as `narrows` and `widens`.

22a ⟨*cases for types indicating opportunities for substitution* 20c⟩+≡ (20b) ◁20c

```
"Div" : addkset(divs, var_for(e.n, varaux, varmap), e)
"Mod" : addkset(mods, var_for(e.n, varaux, varmap), e)
```

To create a balance, we need to have seen both e div k and e mod k. If the variable d is the fresh variable introduced to stand for e, we need to find all variables d that are keys in both `divs` and `mods` with the same k. We then introduce q to stand for the quotient, m to stand for the modulus, and add the balance

$$q = d \operatorname{div} k, m = d \bmod k \bowtie d = k \times q + m,$$

and we equate d to the original dividend (recovered with `varmap`).

22b ⟨B. *use auxiliary tables to add to* `balances`, `neweqns`, *and* `balmap` 21⟩+≡ (20a) ◁21

```
every d := key(divs) & k := key(divs[d]) do
  if ⟨there are any sub-expressions in mods[d, k] ⟩ then {
    qs := divs[d, k]                      # set of all expressions of form Div(d, k)
    q  := var_for(?qs, varaux, varmap)  # fresh var for quotient
    ms := mods[d, k]                      # set of all expressions of form Mod(d, k)
    m  := var_for(?ms, varaux, varmap)  # fresh var for modulus
    insert(balances, balance([balitem(q, Div(d, k)), balitem(m, Mod(d, k))],
                             [balitem(d, k_times_q_plus_m(k, q, m))]))
    add_equation(neweqns, d, varmap[d])
    every balmap[!qs] := q
    every balmap[!ms] := m
  }
```

Finally, we update `balmap` so that in step C the fresh variables q and m will be substituted for all sub-expressions of forms e div k and e mod k.

Because the possibilities of nested and overlapping slices make the slice-balancing code very tricky, and because the other balancing algorithms explain the idea, the slice-balancing code is omitted from this paper.

Introducing balances while the solver runs

One problem with solving equations over the integers is figuring out what to do with equations like $4 \times v = e$, where e is any computable expression. We can't simply divide e by 4, as we would if we were solving over the reals. We do know, however, that 4 must divide e without a remainder, so we can introduce fresh variables d, q, and m, rewrite $4 \times v \mapsto d$ to yield

$$d = e,$$

add the balance

$$q = d \operatorname{div} 4, m = d \bmod 4 \bowtie d = 4 \times q + m,$$

and add equations

$$\begin{array}{rcl} v & = & q \\ m & = & 0. \end{array}$$

Solving this new system gives $v = e \operatorname{div} 4$ with the constraint $e \bmod 4 = 0$.

This code, which the solver runs if it can't otherwise make progress, implements the transformation just described:

```
23a ⟨if a pending expression can be fixed with div or mod, drain pending 23a⟩≡        (12a)
  if ⟨there is an expression z in pending ⟩ &
     ⟨there is a variable v in z that is not an input ⟩ then {
    ⟨remove z from pending ⟩
    ⟨let k be the coefficient of v in z, and make z = k × v ⟩
    if k < 0 then ⟨negate k and z ⟩
    ⟨create fresh variables d, q = d div k, m = d mod k 23b⟩
       # now force d = z, v = q, m = 0
    every put(zeroes, subtract(d, z) | subtract(v, q) | subtract(m, 0))
    ⟨put remaining pending expressions back in zeroes ⟩
  }
```

The "every put(zeroes, ...)" line adds the equations d = z, v = q, and m = 0, and the following code adds the balances:

```
23b ⟨create fresh variables d, q = d div k, m = d mod k 23b⟩≡        (23a)
  ⟨create fresh variables d, q, and m ⟩
  insert(balances, b := balance([balitem(d, k_times_q_plus_m(k, q, m))],
                                [balitem(q, Div(d, k)), balitem(m, Mod(d, k))]))
  ⟨note that the new balance b is associated with d, q, and m ⟩
```

DISCUSSION

Balancing extends the applicability of simple Gaussian elimination by providing a way to eliminate nonlinear operators from otherwise linear equations. The solver based on this method is part of an application generator that encodes and decodes machine instructions (Ramsey and Fernandez 1995). The solver does not compute numerical solutions directly; instead, it computes symbolic solutions, which are used to generate C or Modula-3 code. The generated code does the arithmetic when the application is run. Separating the solving step from the computation of the results yields a very efficient application without an especially efficient solver. By applying the solver twice to a single system of equations, varying the set of variables chosen as inputs, the application generator guarantees consistent encoding and decoding.

It might not be necessary to generate code in a high-level language if you could write equations directly in that language and have them solved. Van Wyk (1992) presents a C++ library that exploits operator overloading to provide an elegant interface to a solver much like that of Derman and Van Wyk (1984). As is, this technique is not suitable for an application like encoding machine instructions, because we don't want to pay the overhead of running the solver and interpreting the answer every time an instruction is encoded. These overheads could be eliminated by introducing "inputs" as described in this paper, and by using run-time code generation to eliminate the interpretation of the answers.

Equations have been used as specifications, both of abstract data types (Guttag and Horning 1993) and of more general computations (Hoffman and O'Donnell 1982). In these kinds of systems, the equations themselves define the meaning of the operators used. Tools that work with such systems work by term rewriting or other kinds of reduction, and they may assume a meaning of equations beyond simple equality, namely, that they define a preferred direction for rewriting (e.g., left to right). In some cases, additional restrictions may be needed to to guarantee that terms can be rewritten into a normal form (Church-Rosser property) or to enable efficient algorithms for rewriting.

By contrast, the solver presented here cannot use equations to define the meanings of operators. Instead, operators must be defined by supplying balancing rules, which include functions used to compute the results of the operators. There are two benefits: the solver can handle some operators that are difficult to define equationally, and the solver imposes no preferred direction on equations. Indeed, the development of the balancing algorithm was motivated by a desire to solve systems of equations in more than one direction. There is also a risk: if the balancing rules for a particular operator are implemented incorrectly, the solver produces wrong answers.

The balancing method cannot handle every nonlinear operator. It requires that variables be divided into two sets, each determining the other, so it cannot help when a majority of variables determine the remainder, as in the two-argument arc-tangent function where

$$\theta = \mathrm{atan2}(x, y),$$

in which, except for special cases, any two of (θ, x, y) determine the third. In other, similar situations, the balance method may be of some help even though it does not exploit all available information. For example, it can convert between polar and rectangular coordinates by using the balance

$$x = r\cos\theta, y = r\sin\theta \bowtie r = \sqrt{x^2 + y^2}, \theta = \mathrm{atan2}(x, y).$$

This ability can be helpful even though the balancer cannot recognize situations in which x and θ determine y and r.

The balancing solver can be useful even when it does not eliminate all nonlinear operators from a system of equations. When nonlinear operators are applied only to inputs or to functions of inputs (i.e., to computable expressions), the solver can generate code to solve for unknown variables. This property has practical value in the machine-code toolkit, making it possible to encode machine instructions that discard parts of their inputs (e.g., by taking only the most significant 16 bits of an operand and ignoring the rest). The HEQS solver (Derman and Van Wyk 1984) also has this property.

Integer linear programming could be used to solve equations containing the three sets of nonlinear operators used to illustrate this paper. Division and modulus can be expressed with the system

$$\begin{array}{rcl} d & = & k \times q + r \\ 0 & \leq & r < k. \end{array}$$

Bit slicing can be expressed by augmenting the linear equation used in balancing with suitable range constraints. Finally, sign extension can be expressed in terms of bit slicing by writing

$$n\!\uparrow_k = n[0{:}k-2] - 2^{k-1} \times n[k-1{:}k-1].$$

The balancing method is worthwhile because it is simple and because it can be applied to different sets of operators.

ACKNOWLEDGMENTS

The solver owes its existence to my collaboration with Mary Fernandez. I thank Preston Briggs, Mike Bianchi, and Referee A for their careful readings of this paper; collectively they found several errors and some related work that I had overlooked. This work was supported in part by Bell Communications Research.

References

Borning, Alan. 1981 (October).
The programming language aspects of ThingLab, a constraint-oriented simulation laboratory.
ACM Transactions on Programming Languages and Systems, 3(4):353–387.

Derman, Emanuel and Christopher Van Wyk. 1984 (December).
A simple equation solver and its application to financial modelling.
Software—Practice & Experience, 14(12):1169–1181.

Griswold, Ralph E. and Madge T. Griswold. 1990.
The Icon Programming Language. Second edition.
Englewood Cliffs, NJ: Prentice Hall.

Guttag, John V. and James J. Horning, editors. 1993.
Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science.
Springer-Verlag.
With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

Hoffman, Cristoph M. and Michel J. O'Donnell. 1982 (January).
Programming with equations.
ACM Transactions on Programming Languages and Systems, 4(1):83–112.

Knuth, Donald E. 1984.
Literate programming.
The Computer Journal, 27(2):97–111.

———. 1986a.
Computers & Typesetting. Volume D, *METAFONT: The Program*.
Addison-Wesley.

———. 1986b.
The Metafontbook.
Reading, MA: Addison-Wesley.

Ladd, David A. and J. Christopher Ramming. 1994 (October).
Two application languages in software production.
In *Proceedings of the USENIX Symposium on Very High Level Languages*, pages 169–177, Santa Fe, NM.

Papadimitriou, Christos H. and Kenneth Steiglitz. 1982.
Combinatorial Optimization: Algorithms and Complexity.
Englewood Cliffs, NJ: Prentice-Hall.

Pugh, William. 1992 (August).
A practical algorithm for exact array dependence analysis.
Communications of the ACM, 35(8):102–114.

Ramsey, Norman and Mary F. Fernandez. 1994 (October).
New Jersey Machine-Code Toolkit architecture specifications.
Technical Report TR-470-94, Department of Computer Science, Princeton University.

———. 1995 (January).
The New Jersey machine-code toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Ramsey, Norman. 1994 (September).
Literate programming simplified.
IEEE Software, 11(5):97–105.

Sutherland, Ivan. 1963.
Sketchpad: A Man-Machine Graphical Communication System.
PhD thesis, MIT, Cambridge, Mass.

Van Wyk, Christopher J. 1992 (June).
Arithmetic equality constraints as C++ statements.
Software—Practice & Experience, 22(6):467–494.