# The Straight-Line Automatic Programming Problem

Rajeev Joshi, Greg Nelson, and Yunhong Zhou
HP Systems Research Center
November 19, 2003

**Abstract**. The paper presents a design for the Denali-2 super-optimizer, which will generate minimum-instruction-length machine code for realistic machine architectures using automatic theorem-proving technology: specifically, using E-graph matching (a technique for pattern matching in the presence of equality information) and boolean satisfiability solving.

The paper presents a precise definition of the underlying automatic programming problem solved by the Denali-2 super-optimizer. It sketches the E-graph matching phase and presents a detailed exposition and proof of correctness of the reduction of the automatic programming problem to the boolean satisfiability problem.

## 1. INTRODUCTION

The Automatic Programming Problem is the problem of automatically finding a program that meets a given specification. For example, the problem of finding a C program that meets a specification given by a precondition and a postcondition is an instance of the automatic programming problem. The "planning problems" of Artificial Intelligence are also instances of the automatic programming problem.

In its full generality the problem is of high computational complexity in the worst case, comparable to the decision problems of mathematical logic. But the difficulty of the problem is matched by its importance, and it has many special cases that are worthy of study. In this paper, we define and solve a special case in which the program to be found is restricted to be a sequential composition of primitives that model machine instructions, and in which the specification that must be met is a multi-assignment (a sequence of expressions to be computed together with a sequence of locations into which the values of the expressions are to be stored). We call this special case the "straight-line automatic programming problem". The practical significance of our solution is that it applies to the engineering problem of generating optimal machine code for modern computing architectures, which was all but abandoned when it was found in the 1970's to be NP-hard.

Some writers define a problem to be "tractable" if it admits a polynomial-time solution and "intractable" otherwise. The authors of this paper think that it is a terminological mistake to give a technical meaning to a word with such a strong non-technical meaning, since a word so defined cannot be used in its technical sense without conveying at least the connotation of its ordinary sense. Readers who like the terminology might ask themselves what they would think of defining a problem to be "significant" if it has no polynomial time solution and "insignificant" if it does.

Terminology aside, the straight-line automatic programming problem certainly cannot be solved in polynomial time in the worst case, since such a solution would provide an algorithmic solution to the decision problem for first-order logic, and therefore to the halting problem. Our solution reduces the problem to the first-order satisfiability problem, which is also difficult in the worst case, but for which many heuristics have been found in practice. In other words, we apply automatic theorem-proving methods to the problem of generating optimal code. The two methods that are used in our solution are E-graph matching and boolean satisfiability solving. This paper sketches the role of matching, and focuses on the reduction of the straight-line automatic programming problem to the boolean satisfiability problem.

The context of our work is a practical engineering effort: the Denali-2 super-optimizing code generator project at the former HP Systems Research Center, which generates optimal machine code for modern architectures. (Since in the context of code generation, the word "optimizer" is universally used with the meaning of a code improver, we follow Massalin [5] in using the word *super-optimizer* for an optimizing code generator that attempts to find a true optimum by some criterion.) The Denali project's first prototype, Denali-1, generates cycle-optimal code for the EV6 implementation of the Alpha architecture. Our current project, Denali-2, is designed to generate optimal code for upcoming implementations of the Intel IA-64. Denali-1 was described in a recent PLDI paper [4], which will hereafter be cited as "the Denali-1 paper". The Denali-1 paper was written in a style that aimed to acquaint the computer architecture and compiler research communities with a new kind of code generator, with examples and high-level prose expositions of the major algorithms and data structures. This paper, in contrast, is written in a more precise style, with definitions and theorems and proofs. The reason for the shift of style is that as we started work on Denali-2, we found that we needed more precision to keep our programming project from sliding into complexity and becoming a mess. This paper has been written for publication, but it is also a working engineering project document. We felt that our algorithm was sufficiently simple and self-contained to publish, and that writing first and coding second was likely to produce a better result than the other way around.

It seems unfair to submit the reader to our definitions and theorems without even a scrap of evidence that the Denali method can be of engineering value, so we provide this one anecdote, which is all we have space for: our Denali-1 prototype found a twenty instruction, five cycle program for byte-swapping a four byte value (see figure 5 of the Denali-1 paper). Since its output was cycle-optimal by design, we were surprised to find that it contained an instruction whose output was overwritten by the following instruction! Investigation revealed the reason: an early stage of the EV-6 pipeline fetches a group of four instructions and enqueues each of them for one of the four ALUs on the chip. The assignment of instructions to queues uses a heuristic based on the vector of four op-code types of the fetched instructions. Once an instruction is placed in a queue, if a different queue empties before the one chosen by the heuristic, the chip will not transfer the instruction, and one or more cycles may be wasted. In the case of the byte-swap code, eliminating the unused instruction (or replacing it with a no-op) leads to one of the uncommon situations in which the heuristic wastes cycles. Inserting the unusued instruction changes the vector of four opcodes in a way that causes the heuristic to get a better answer, and more than makes up for the unused instruction. Satisfiability solving may be "intractable" by definition, but this example shows that a satisfiability solver can produce a solution that in a human being would be considered clever, at least. The prototype takes just over a minute to generate this sequence, during which time it solves four satisfiability problems, the largest of which contains several thousand variables and several thousand clauses. We are confident that a machine language programmer of ordinary experience would never come up with the code, and that an EV-6 machine language guru would not find it as rapidly as Denali.

Because this paper is more formal and precise than the Denali-1 paper, we have more confidence in the correctness of our algorithm in this paper than we do in the correctness of Denali-1's implementation. But Denali-1 is an implemented prototype, while this paper is just an algorithm, that is, a paper design. Also, this paper makes two simplifications that were not made in Denali-1. The first simplification is that while the inner subroutine of Denali-1 generates code for a guarded multi-assignment, our algorithm generates code for an ordinary multi-assignment. This simplification is rather minor, and we are confident that our algorithm could be modified to handle guarded multi-assignments with no serious difficulty. The second simplification is that while Denali-1 produces code that is optimal in the sense of minimizing the number of cycles, our algorithm generates code that is minimal only in the number of instructions, not in the number of cycles. This simplification is rather major, and although we think that our result is a useful stepping stone to the cycle-minimization problem that we hope to work on next, at this point we don't really know.

Another possible metric for optimality is the number of extra registers used by a computation. Perhaps the Denali method could be modified to find code that minimizes the number of extra registers, but we won't try to do so in this paper. In fact, to simplify our instruction-minimizing solution, we will make the assumption that there is no shortage of registers for holding intermediate results. Of course this assumption simplifies our algorithm, and the machine language gurus to whom we aim to offer our tool tell us that the assumption is appropriate for modern architectures.

## 2. STATEMENT OF THE PROBLEM TO BE SOLVED

This section presents a number of definitions, building up to a precise statement of the problem to be solved.

### 2.1 S-expressions

Denali's input is encoded as a sequence of *S-expressions*, where an S-expression is either an *atomic symbol* or a parenthesized list of S-expressions, an atomic symbol is either a numeral, a quoted string, or an *identifier*, and an identifier is either alphanumeric (like "`mem`" and "`r6`") or symbolic (like "`+`", "`*`", and "`**`").

### 2.2 Components

The coordinate functions of the state space acted upon by a program are often called *variables*. They are also sometimes called *components*, and that is what we will call them in this paper. An instance of the straight-line automatic programming problem includes a list of the components that the solution program is to act upon. In the code generation application, this is simply the list of hardware components of the target architecture. These components are partitioned into *registers* and *non-registers*.

Typically the non-registers include a memory, but there is nothing in our algorithm that requires this. In our examples we will assume the components include a memory named `mem` and registers named `r0`, `r1`, . . . .

### 2.3 Machine operations and mathematical functions

Atomic symbols that denote functions are *function symbols*. Some, like `add64` (addition modulo $2^{64}$), denote functions that the machine's ALU can compute, these are called *machine operations*. Others denote functions that are not computable by the ALU, they are called *mathematical functions*. Examples of mathematical functions are $+$, which in Denali denotes mathematical addition, and `**`, which denotes exponentiation.

Two more examples of function symbols are `rd` and `wr`, where `(rd a i)` is element `i` of the array `a`, and `(wr a i x)` is the array that has the same elements as array `a` except for element `i`, which has the value `x`.

Associated with each function symbol is a natural number called its *degree*, which is simply the number of arguments to the function. For example, `rd` and `add64` are of degree two, and `wr` is of degree three.

### 2.4 Terms

A *term* is either (a) the name of a component, (b) a numeral, or (c) an expression of the form $f(t_0, \ldots, t_{m-1})$ where $f$ is a function symbol of degree $m$ and the $t$'s are terms. In a term of the form (c), $f$ is the *operation* and the $t_j$'s are the *arguments*. The *depth* of a term of the form (a) or (b) is zero, and the depth of a term of the form (c) is one plus the maximum depth of any $t_j$. Denali represents terms as S-expressions in the standard way (for example `(add64 n m)` denotes the application of the function `add64` to the values of `n` and `m`), but in this paper we will sometimes take the liberty of using conventional mathematical notation, for example, writing $f(x)$ instead of `(f x)`, or even $t + u$ instead of `(+ t u)`.

### 2.5 Multi-assignments and updates

A *multi-assignment* is an expression of the form

$$(c_1, \ldots, c_n) := (t_1, \ldots, t_n)$$

where the $c$'s are (the names of) distinct components and the $t$'s are terms. We remind the reader of the multi-assignment's meaning, operational and semantic, as expounded in Dijkstra's classic text [2]: Operationally, a multi-assignment specifies that the $t$'s be evaluated and that each $t_i$'s value be stored in $c_i$. All the evaluations are performed before any of the updates. Semantically, the multi-assignment is a substitution of the $t$'s for the $c$'s to be performed on a postcondition to produce the corresponding weakest precondition. Thus the substitutions studied in mathematical logic are in fact the assignment statements used by programmers. In this paper we call each constituent pair $c_i := t_i$ of a multi-assignment an *update*; $c_i$ is the *target* of the update and $t_i$ is the *term* of the update. Many programming languages impose the requirement $n = 1$ on a multi-assignments, that is, they require that each multi-assignment consist of a single update, but no treatment of mathematical logic that we know of has ever imposed such a requirement on substitutions, and we certainly won't do so here.

The set of multi-assignments is semantically closed under sequential composition. For example, $x := a$; $(y, z) := (x, x)$ is semantically equivalent to $(x, y, z) := (a, a, a)$.

To encode a multi-assignment, Denali uses a list whose first element is the assignment symbol and whose remaining elements are the constituent updates of the multi-assignment, where each update is encoded as a two-element list whose first element is the update target and whose second element is the update term. For example,

```
(:= (x (+ x y)) (y x))
```

encodes

$$(x, y) := (x + y, x) \quad .$$

The order of the updates in the encoding is insignificant.

### 2.6 Axioms

An axiom is a first-order formula that specifies a property of the function symbols. Denali's axiom syntax allows quantifiers, boolean connectives, function symbols and the equality relation. The details of the axiom syntax are not important in this paper and will not be described, but only suggested by examples.

Here are some example axioms:

```
(FORALL (x y) (= (+ x y) (+ y x)))
```

That is, addition is commutative.

```
(FORALL (a i x) (= (rd (wr a i x) i) x))
```

That is, if you write array element $i$ and then read it, you read what you wrote.

```
(FORALL (a i j x)
    (OR (= i j)
        (= rd (wr a i x) j) (rd a j)))
```

That is, writing array element $i$ doesn't change any other element $j$.

### 2.7 Templates

A *template* is a parametrized pattern whose instances are instructions. Syntactically, a template is an expression of the form

```
(TEMPLATE <paramtypes> <semantics> <syntax>)
```

where `<paramtypes>` is a parenthesized list of parameter types, `<semantics>` is a multi-assignment, the semantics of the instruction, and `<syntax>` is a string, the syntax for the instruction (in the case of Denali-2, it is the assembly language syntax).

In this paper, we will consider only two parameter types: `reg`, which denotes a register, and `immediate`, which denotes an integer value encoded within the instruction itself. In practice there are likely to be different types of registers and immediates of different bit sizes, but we think these considerations can be addressed by straightforward modifications to the paper.

Within `<semantics>` and `<syntax>`, parameter number $k$ (counting from zero) is denoted by the expression `#k`.

An instance of the template is obtained by choosing an actual register for each formal parameter of type `reg` and an actual numeral for each formal parameter of type `immediate` and substituting the actuals for the formals in `<semantics>` to get the instruction semantics and performing the same substitution in `<syntax>` to get the instruction syntax. To be legal, the substitution of actuals for formals on `<semantics>` must produce a legal multi-assignment, that is, a multi-assignment in which the target components are distinct.

For example, here are templates for add, load, and store instructions in a hypothetical architecture that uses an assembly language syntax somewhat like the standard for the IA-64:

```
(TEMPLATE (reg reg reg)
          (:= (#2 (add64 #1 #0)))
          "add #2= #1, #0")

(TEMPLATE (reg reg)
          (:= #1 (rd mem #0))
          "load #1= #0")
```

```
(TEMPLATE (reg reg)
          (:= (mem (wr mem #0 #1)))
          "store #0= #1")
```

Continuing the example: The instance of the third template via the substitution (`#0`, `#1`) := (`r0`, `r1`) is the instruction with syntax

```
    store r0= r1
```

and semantics

```
    (:= (mem (wr mem r0 r1)))
```

That is, an instruction that stores register `r1`'s value into the memory cell addressed by register `r0`.

### 2.8 Restrictions on the templates

An instance of the straight-line automatic programming problem includes a list of templates, which specify the instructions that are available. We impose two restrictions on these templates: the depth-one rule and the unique occurrence rule. The utility of these rules will become apparent in Section 2.8.2.

**The depth-one rule.** Each term on the right side of the multi-assignment `<semantics>` of a template must be of depth one, that is, consist of the application of a function symbol to a list of atoms (components, numerals, or template parameters).

**The unique occurrence rule.** Each machine operation appears exactly once among the templates (as the operation of the term of one of the updates of the template's semantics).

2.8.1 *The rules are not as strict as they seem.* These two rules may seem overly strict, but in the presence of axioms they don't actually rule out any useful possibilities. Here are two examples that illustrate how the rules can be troublesome and how axioms can circumvent the trouble: an indexed load instruction whose template might have been

```
(TEMPLATE
   (reg immediate reg)
   (:= (#2 (rd mem (add64 #0 #1))))
   "load #2= #0(#1)")
```

and an auto-incrementing load instruction, whose template might have been

```
(TEMPLATE
  (reg reg)
  (:= (#1 (rd mem #0)) (#0 (add64 #0 1)))
  "load #1= #0, 1")
```

The first template violates the depth-one rule. And if either template is used, the unique occurrence rule will prohibit `add64` from being used anywhere else in the templates. (And we are bound to want to use it in the template for the `add` instruction!)

But since the input to Denali includes axioms, we simply define new function symbols to work around these problems:

```
(FORALL (a n m) (= (iload a n m) (rd a (add64 n m))))
(TEMPLATE (reg immediate reg)
```

```
            (:= (#2 (iload mem #0 #1)))
            "load #2= #0(#1)")

(FORALL (n) (= (autoinc n) (add64 n 1)))
(TEMPLATE
  (reg reg)
  (:= (#1 (rd mem #0)) (#0 (autoinc #0)))
  "load #1= #0, 1")
```

So these rules don't prevent us from specifying useful architectures with the templates.

A consequence of the depth-one rule is that we cannot use what is perhaps the most obvious template for the `mov` instruction. Instead, therefore, our plan is to introduce a function symbol `id`, to assert $\mathtt{id}(q) = q$ for each *scalar* equivalence class $q$, where all classes are scalar except those of `mem` and those containing `(wr a i x)` for non-scalar `a`, and then to use the rule-abiding template

```
(TEMPLATE (reg reg) (:= #1 (id #0)) "mov #1= #0")
```

2.8.2 *Three notations.* We define three pieces of notation that depend on the unique occurrence rule and the depth-one rule:

For any machine operation $f$:

*template*$(f)$ denotes the template in which $f$ occurs,

*target*$(f)$ denotes the target of the update containing the occurrence of $f$ in *template*$(f)$, and

$src_j(f)$ denotes the component, numeral, or template parameter that appears as argument $j$ (counting from zero) of the occurrence of $f$ in *template*$(f)$.

In summary, for any machine op $f$, the semantics of *template*$(f)$ includes the update

$$target(f) := f(src_0(f), \ldots, src_{m-1}(f)) \quad ,$$

where $m$ is the degree of $f$.

This is a convenient place for one further definition: If *target*$(f)$ is a component $c$ (not a template parameter) then $f$ is *output-restricted to* $c$. In the architecture given by our example templates (as in all conventional architectures), `wr` is output-restricted to `mem`.

2.8.3 *Argument types.* It will be important to classify the different types of arguments to a machine operation:

*In-place arguments.* If $src_j(f)$ and *target*$(f)$ are both `#k`, for some template parameter `#k` of type `reg`, then argument $j$ to $f$ is an *in-place* argument. In the architecture given by our example templates, the argument to `autoinc` is an in-place argument.

*Source-restricted arguments.* If $src_j(f)$ is a component $c$ that is not a template parameter, then argument $j$ of $f$ is *source-restricted to* $c$. In the architecture given by our example templates (as in all conventional architectures), `rd` and `wr` both have an argument that is source-restricted to `mem`. These are the only source-restricted arguments in our examples, but the Denali solution to the automatic programming problem allows arguments that are source-restricted to registers. Indeed, this flexibility is essential to Denali-2's handling of procedure calls, since calling standards require that procedure arguments be taken from particular registers. But we will not go further into this subject in this paper.

*Free-source arguments.* If $src_j(f)$ is a register parameter `#k` and *target*$(f)$ is either a fixed component or a register parameter other than `#k`, then argument $j$ of $f$ is a *free-source* argument. In the architecture given by our example templates, every argument that is not an in-place argument or source-restricted to `mem` is a free-source argument.

*Immediate arguments.* If $src_j(f)$ is a template parameter of type `immediate`, then argument $j$ to $f$ is an *immediate* argument.

*Constant arguments.* If $src_j(f)$ is a numeral, then argument $j$ of $f$ is a *constant* argument. These are not common, and our example templates contain none of them.

We remark that these argument types are exhaustive and mutually exclusive.

### 2.9 Definition of the problem to be solved

A program $P$ *implements* a multi-assignment $M$ *using temporaries* $L$, where $L$ is a list of components, if executing $P$ causes each of $M$'s updates to be performed and modifies no component except the targets of $M$ and the temporaries of $L$.

The input to the *straight-line automatic programming problem* is a list of components, a collection of instruction templates satisfying the unique occurrence and depth-one rules, a collection of axioms constraining the interpretation of the operators in the templates, a *budget* $N \geq 0$, and a multi-assignment $M$ called the *goal*. The output is either a sequential composition

$$P \equiv I_0; \cdots; I_{N-1}$$

of $N$ instructions, each of which is an instance of one of the templates, together with a list $L$ of temporary registers, such that $P$ implements $M$ using the temporaries $L$, or a proof that no such solution $(P, L)$ exists.

Given the ability to solve this problem, minimum-instruction code is generated with a simple search loop that tries various budgets until, for some $K$, a program of $K$ instructions is found together with a proof that no program of $K - 1$ instructions exists.

## 3. THE DENALI SOLUTION

The Denali solution to the straight-line automatic programming problem has two phases: matching and planning. The matching phase builds a data structure called an E-graph. The E-graph is a data structure that represents all possible ways of computing the terms of the goal multi-assignment. The planning phase takes the E-graph and generates a CNF satisfiability problem, which it passes to a satisfiability solver. Finally, it transforms the satisfiability solution, if one is found, into a solution of the straight-line automatic programming problem.

### 3.1 E-graphs

The main part of this paper is about the planning phase, not the matching phase, and therefore we will not describe the implementation of the matching phase. But we will briefly describe the E-graph that the matcher constructs, which is a crucial input to the planning phase.

An E-graph is a term DAG together with an equivalence relation on its nodes. The equivalence relation relates two nodes if the corresponding terms are semantically equivalent (certain to be equal in the context of the program to be generated). The terms in the DAG include all the terms in the goal multi-assignment, and all semantically equivalent terms that can be computed by the target architecture. For example, if either of the nodes for `r1+r2` and `r2+r1` are present, then both will be present, and they will be equivalent. Two nodes with the same operator and the same number of children, say $\mathtt{op}(t_1, \ldots, t_m)$ and $\mathtt{op}(u_1, \ldots, u_m)$, are *congruent* if, for each $i$, the node for $t_i$ is equivalent to the node for $u_i$. The equivalence relation is *congruence-closed* if any two congruent nodes are equivalent. Since equality is a congruence, Denali always keeps the equivalence relation in the E-graph congruence-closed. For this purpose, Denali uses the efficient congruence closure algorithm described by Downey, Sethi and Tarjan [3].

All congruent nodes are equivalent, but not all equivalent nodes are congruent. For example, an E-graph equivalence class might include the two equivalent, non-congruent nodes $n * 2$ and $n + n$. But any nodes of the form $\mathtt{op}(n * 2)$ and $\mathtt{op}(n + n)$, if they exist, will be both congruent and equivalent. When dealing with an E-graph instead of an ordinary term DAG, it is best to think of a term as being represented by a class of congruent nodes, whose general form is $\mathtt{op}(q_0, \ldots, q_{m-1})$, where the $q$'s are equivalence classes of nodes. In this paper, we sometimes write "term" when it would be more precise to write "congruence class".

Denali initializes its E-graph to the term DAG containing all the terms of the updates of the goal multi-assignment, with the identity equivalence relation, and uses the axioms to introduce into the E-graph additional terms and equivalences between terms until eventually producing an E-graph that represents all possible ways of computing the terms of the goal multi-assignment on the target architecture. For an illustrative example of the matching phase, see the Denali-1 paper. For a detailed account of the implementation of the matcher, see the comprehensive paper about the theorem prover Simplify [1].

For the remainder of this paper, we simply assume that an appropriate E-graph has been constructed by the matcher.

For a term $t$, we write $\mathit{class}(t)$ to denote the equivalence class of $t$ in the E-graph.

## 3.2 The planning phase

We now come to the core of this paper, which presents a detailed design for the planning phase of Denali-2.

Given an E-graph that represents all possible ways of computing the goal terms, the planning phase finds an explicit sequence of machine instructions that computes them and stores their values in the targets of the goal multi-assignment. For each equivalence class that must be computed, the planning phase must choose a particular term in the class whose operation can be computed by the target architecture. The planning phase must also pack multiple updates into instructions as allowed by the architecture and as necessary to compute the goal within the budget. Denali's planning phase takes as input an E-graph, a goal multi-assignment, a set of templates, and a budget. It creates a collection of boolean unknowns, generates boolean satisfiability constraints on the unknowns that are satisfiable exactly when the instance of the straight-line automatic programming problem is solvable and uses an off-the-shelf satisfiability solver to determine whether these constraints are satisfiable.

Before we describe the unknowns and the constraints on them, we present one lemma, the "Destination Lemma", which limits the search for an instruction sequence by exploiting the homogeneity of the register architecture.

The lemma requires four preliminaries:

First, a sequential composition of multi-assignments $I_1; \ldots; I_n$ *computes the term $t$ into* *the component* $c$ if the semantics of the sequential composition includes the update $c := t$, and it computes a class $Q$ into a component $c$ if it computes any term in $Q$ into $c$. For example, if instruction $I$ has semantics `r1` $:=$ `r1`$+1$, then the sequence `r1` $:=$ `r6`; $I$; $I$; $I$ computes `r6` $+ 1 + 1 + 1$ into `r1`.

Second, an equivalence class $Q$ is *c-homogeneous*, for a component $c$, if every element of $Q$ is either the term $c$, or a term that is the application of an operation that is output-restricted to $c$. This is a rather rare property except when $c$ is `mem`, in which case essentially all non-scalar equivalence classes can be expected to be `mem`-homogeneous.

Third, now and for the rest of the paper we assume that, for each scalar equivalence class $Q$ of the E-graph, a register *temp*$(Q)$ has been selected, such that if $Q$ and $R$ are distinct classes, then *temp*$(Q)$ and *temp*$(R)$ are distinct registers. In case it is necessary to compute $Q$ into a temporary register, *temp*$(Q)$ is the register that will be used.

Fourth, for each class $Q$, we define a set of components *dest*$(Q)$ as follows: If $Q$ is $c$-homogeneous for some $c$, then *dest*(Q) is the singleton $\{c\}$. Otherwise, *dest*$(Q)$ is defined by the rule that a component $c$ is in *dest*$(Q)$ if any of the following four conditions hold

(1) for some term $t$ in $Q$, $c := t$ is an update of the goal multi-assignment, or
(2) $Q$ contains some term whose operation is output-restricted to $c$, or
(3) The E-graph contains some term

$$g(q_0, \ldots, q_{j-1}, Q, q_{j+1}, \ldots, q_{m-1})$$

where argument $j$ of $g$ is source-restricted to $c$, or
(4) $c$ is the component *temp*$(Q)$.

The point of this definition is that when considering components into which $q$'s value might usefully be computed, *dest*$(q)$ is the set of candidates.

**The Destination Lemma.** If an instance of the straight-line automatic programming problem has a solution with $n$ instructions, then it also has a solution with $n$ instructions that never computes any class $Q$ into any component outside *dest*$(Q)$.

Consider a solution that computes a term $T$ into some component $d$ not in *dest*(*class*$(T)$). It cannot be that *class*$(T)$ is $c$-homogeneous, since terms in such a class can be computed only into $c$, so and in this case *dest*(*class*$(T)$) is $\{c\}$. So it must be that *dest*(*class*$(T)$) is defined by the four-condition rule. We transform the solution into an equal-length solution that doesn't compute $T$ into a component outside *dest*(*class*$(T)$), by substituting *temp*(*class*$(T)$) in the instruction sequence for the component $d$. For the instruction that computed $T$ into $d$, condition (2) implies that we can compute $T$ into *temp*(*class*$(T)$) instead, without violating the architecture. (If $T$ were the application of an output-restricted function, then $d$ would be in *dest*(*class*$(T)$) contrary to supposition, so *target*$(f)$, where $f$ is the root operator of $T$, must be a register parameter, and can be instantiated to *temp*(*class*$(T)$) instead of to $d$.) For the later instructions that use the value in $d$, condition (3) implies that we can use *temp*(*class*$(T)$) as a source instead of $d$, without violating the architecture. Condition (1) implies that $d$ was used only as a temporary, not as a goal, so the new sequence is as good as the old. Condition (4) implies that the new sequence meets the limitation.

In other words, the lemma says that limiting ourselves to code sequences that compute *class*$(T)$ into locations of *dest*(*class*$(T)$) only cannot cause us to miss an optimal instruction sequence. Since the average size of *dest*$(Q)$ is small (most commonly of size one or two), this drastically reduces the search space.

Denali doesn't choose actual registers *temp*$(Q)$ in advance. There are not enough registers to do this. Instead, Denali simply treats each syntactic expression *temp*$(Q)$ as a "virtual register". After the code is generated, any of these virtual registers that are actually used by the code are replaced by physical registers, and care is taken to assign distinct physical registers to any two virtual registers whose lifetimes overlap in the generated code.

We now describe the Denali reduction of the planning portion of the straight-line automatic programming problem to the boolean satisfiability problem. First we introduce the unknowns, then the constraints on the unknowns, then show how to construct a program from a solution to the constraints.

3.2.1 *Unknowns.* For each instruction index $i$ in the range $0 \le i < N$, each E-graph congruence class $t$, and each component $c$ in *dest*(*class*$(t)$), we introduce the unknown

$$L(i, t, c)$$

which will be constrained to be true if one of the updates of instruction $i$ computes the value of $t$ into $c$ by applying the operator of $t$ to a vector of arguments that contain the values of the arguments of $t$.

For each $i$ in the range $0 \le i \le N$, that is, each length of any prefix of the instruction sequence, including the improper prefix consisting of all $N$ instructions, each equivalence class $q$, and each component $c$ in *dest*$(q)$, we introduce the unknown

$$H(i, q, c)$$

which will be constrained to be true if the value of equivalence class $q$ is held in component $c$ in the state produced by executing the first $i$ instructions of the generated code.

For each instruction index $i$ in the range $0 \le i < N$ and each template *tmpl*, we introduce the unknown

$$U(i, tmpl)$$

which will be constrained to be true if instruction $i$ is an instance of template *tmpl*.

For each instruction index $i$, template *tmpl*, register parameter `#j` of *tmpl*, and for some but not all registers $c$, we introduce the unknown

$$B(i, tmpl, \#\mathtt{j}, c)$$

which will be constrained to be true if instruction $i$ is an instance of *tmpl* by a binding that binds `#j` to $c$.

For each instruction index $i$, template *tmpl*, immediate parameter `#j` of *tmpl*, and for some but not all numerals $n$ appearing in the E-graph, we introduce the unknown

$$B(i, tmpl, \#\mathtt{j}, n)$$

which will be constrained to be true if instruction $i$ is an instance of *tmpl* by a binding that binds `#j` to $n$.

The $c$'s and $n$'s for which $B$-unknowns are required will be explained in Section 3.2.2 (in the paragraphs on the Argument and Result Constraints).

3.2.2 *Constraints.* In stating the constraints, we will use Dijkstra's notation

(`op` `dummy` : `range` : `term`)

to indicate the combination via the operation `op` of the values assumed by the term `term` as the dummy `dummy` ranges over the range `range`. If the range is obvious from the context, it may be omitted.

**The Initialization constraint.** The initialization constraint is

$$(\texttt{AND}\quad c, q :: H(0, q, c) \Rightarrow c \in q) \quad ,$$

where $c$ ranges over all components and $q$ ranges over all equivalence classes. This is a conjunction of constraints each of which is of one of the two forms $H(0, \ldots, \ldots) \Rightarrow$ `TRUE` or $H(0, \ldots, \ldots) \Rightarrow$ `FALSE`. Those of the first form are vacuous and can be omitted; those of the second form are negative unit clauses that prevent the satisfiability solver from "solving" the code generation problem by postulating that useful updates have already been performed in the initial state.

**The Completion constraint.** The completion constraint is that, for each update $c := t$ of the goal multi-assignment,

$$H(N, \mathit{class}(t), c) \quad .$$

That is, the code performs each goal update within the cycle budget.

**The Argument Constraints.** The argument constraints ensure that if the code includes an instruction to compute `op(args)`, it must previously have computed `args` into a vector of components whence the architecture allows the arguments of `op` to be fetched. Also, these constraints ensure that the $B$ unknowns are set to reflect the template instantiations needed to produce instructions that read the arguments from the proper components.

There is one argument constraint for each edge in the E-graph, and its form depends on the type of the argument represented by the edge, as follows.

For each E-graph congruence class $f(q_0, \ldots, q_{m-1})$, for each $j$ such that argument $j$ to $f$ is an in-place argument, for each $c$ in $\mathit{dest}(\mathit{class}(f(q_0, \ldots, q_{m-1})))$, create the unknown $B(i, \mathit{template}(f), \mathit{src}_j(f), c)$ and constrain:

$$\begin{array}{l} L(i, f(q_0, \ldots, q_{m-1}), c) \Rightarrow \\ \quad H(i, q_j, c) \ \texttt{AND}\ B(i, \mathit{template}(f), \mathit{src}_j(f), c) \end{array}$$

That is, to compute $f(q_0, \ldots, q_{m-1})$ into $c$ in instruction $i$, an in-place argument $q_j$ must previously have been computed into $c$, and the template parameter that is both $\mathit{src}_j(f)$ and $\mathit{target}(f)$ must be instantiated to $c$ in instruction $i$.

For each E-graph congruence class $t \equiv f(q_0, \ldots, q_{m-1})$, for each $j$ such that argument $j$ to $f$ is a free-source argument, for each $d$ in $\mathit{dest}(\mathit{class}(t))$, create the unknown $B(i, \mathit{template}(f), \mathit{src}_j(f), d)$ and constrain:

$$\begin{array}{l} L(i, f(q_0, \ldots, q_{m-1}), c) \Rightarrow \\ \quad (\texttt{OR}\ \ d : d \in \mathit{dest}(q_j) : H(i, q_j, d)\ \ \texttt{AND}\ \ B(i, \mathit{template}(f), \mathit{src}_j(f), d)) \end{array}$$

That is, free-source arguments must be computed before the operation that uses them, but they can be computed into any convenient register. Without the destination lemma, the width of the disjunction over $d$ would be equal to the number of registers, which would be unacceptably wide. With the optimization, we expect the average width to be just above one. Frequently $d$ will range over the singleton $\{\mathit{temp}(q_j)\}$; in case $q_j$'s value is a procedure argument or a goal update term (or both), the range will include a second (or third) component.

For each instruction index $i$, each E-graph congruence class $f(q_0, \ldots, q_{m-1})$, for each $j$ such that argument $j$ to $f$ is source-restricted, constrain:

$$L(i, f(q_0, \ldots, q_{m-1}), c) \Rightarrow H(i, q_j, src_j(f))$$

That is, an argument that is source-restricted to $c$ must be held in $c$ before it can be used. If argument $j$ to $f$ is source-restricted, then $src_j(f)$ is not a template parameter, so in this case there is no need to create and constrain a $B$ unknown.

For each instruction index $i$, each congruence class $t \equiv f(q_0, \ldots, q_{m-1})$, each $j$ such that argument $j$ to $f$ is an immediate argument, let $n$ be the numeral in $q_j$, create the unknown $B(i, \mathit{template}(f), src_j(f), n)$, and constrain:

$$L(i, f(q_0, \ldots, q_{m-1}), c) \Rightarrow B(i, \mathit{template}(f), src_j(f), n)$$

That is, an immediate argument can be any compile-time-constant natural number. In this case, if $q_j$ holds no numeral, then constrain $L(i, f(q_0, \ldots, q_{m-1}), c)$ to be `FALSE`.

For each instruction index $i$, each congruence class $t \equiv f(q_0, \ldots, q_{m-1})$, each $j$ such that argument $j$ to $f$ is a constant argument, constrain:

$$L(i, f(q_0, \ldots, q_{m-1}), c) \Rightarrow src_j(f) \in q_j \quad .$$

Each of these constraints is of the form $L(...) \Rightarrow$ `TRUE` or $L(...) \Rightarrow$ `FALSE`. Those of the first form are vacuous and can be omitted. Those of the second form are negative unit clauses that prevent the computation of $f(q_0, \ldots, q_{m-1})$ if some argument $j$ of $f$ is a constant argument that isn't the value of $q_j$.

**The result constraint.** For each instruction index $i$, each term $t$ of the form $f(q_0, \ldots, q_{m-1})$ such that $\mathit{target}(f)$ is a register parameter `#k`, and each register $c$ in $\mathit{dest}(\mathit{class}(t))$, create the unknown $B(i, \mathit{template}(f), \texttt{\#k}, c)$ and constrain:

$$L(i, t, c) \Rightarrow B(i, \mathit{template}(f), \texttt{\#k}, c) \quad .$$

That is, to compute $f(\ldots)$ into $c$ in instruction $i$, the template parameter that represents the register into which the instruction stores $f(\ldots)$ must be instantiated to $c$.

**The two-case hold constraint.** For each $i$ in the range $0 \le i \le N$, each equivalence class $q$, and each component $c$ in $\mathit{dest}(q)$, constrain:

$$\begin{array}{l} H(i, q, c) \Rightarrow \\ \qquad (\texttt{OR} \;\; t : t \in q : L(i-1, t, c)) \\ \quad \texttt{OR} \;\; (H(i-1, q, c) \;\texttt{AND}\; (\texttt{AND} \;\; u :: \neg L(i-1, u, c))) \end{array}$$

That is, a component holds the value of a class after instruction $i-1$ only if that instruction computed one of the class's terms into the component, or the component already held the required value before the instruction and the instruction did not overwrite the value. In the second conjunct of the second disjunct of the consequent of the implication, the dummy $u$ ranges over all terms of the E-graph (although, without change of meaning, we can omit the terms in $q$).

**The issue constraints.** The issue constraints guarantee that for each $i$, the set of $L(i, t, c)$ variables that are assigned true by the satisfiability solver specify a collection of updates that actually can all be performed by some instruction allowed by the templates. They connect the $L$ unknowns to the $U$ and $B$ unknowns.

The first part of the issue constraint is that, for each instruction index $i$, each congruence class $t \equiv f(q_0, \ldots, q_{m-1})$, and each component $c$ in *dest*(*class*($t$))):

$$L(i, t, c) \Rightarrow U(i, template(f))) \quad .$$

That is, an instruction can apply an operation only if it is an instance of the template of the operation.

The second part of the issue constraint is:

$$(\texttt{AND} \ \ i :: (atMost1\ tmpl :: U(i, tmpl)))$$

in which $i$ ranges over instruction indexes, *tmpl* ranges over all templates, and in which we use the notation (*atMost1* `dummy` : `range` : `term`) to specify that as the dummy `dummy` ranges over the range `range`, the boolean term `term` is true at most once. That is, each instruction must be an instance of one single template.

It would take quadratically many clauses to encode *atMost1* exactly in CNF, but a positive occurence can be encoded in linearly many clauses by introducing linearly many auxiliary variables. We will take care to use the construct in positive positions only.

The third part of the issue constraint is

$$(\texttt{AND} \ \ i, tmpl, k :: (atMost1\ x :: B(i, tmpl, \#k, x)))$$

where $i$ ranges over instruction indexes and *tmpl*, $k$, and $x$ range over all values such that $B(i, tmpl, \#k, x)$ is defined.

That is, in any one template instantiation, no formal parameter can be bound to two different actual parameters. The solver cannot satisfy this constraint by falsifying all the $B$ unknowns, because the appropriate case of the argument constraint forced the appropriate $B$ unknown to be true when an operation was applied to an argument, and the appropriate case of the result constraint forced one of the $B$ unknowns to be true when a result was computed by an instruction into a register.

3.2.3 *Constructing the program from a satisfiability solution.* We now describe how to construct a program from a solution to the boolean satisfiability constraints listed above.

Let $s$ be a solution to the constraints. We write $s \models b$ to indicate that the boolean condition $b$ is true in the solution $s$. We construct a program from $s$ with a simple loop.

The loop relies on the procedure *GetSource*, where *GetSource*($i$, $j$, $t$, $c$) finds and returns the *source* of argument $j$ to the operation of the term $t$ whose value will be stored into $c$ as one of the updates of instruction $i$ of the solution. The source of an immediate argument is the numeral denoting the argument's value, the source of any other kind of argument is the component that holds the argument's value in the pre-state of instruction $i$. A more precise specification and implementation for *GetSource* are presented in Figure 1.

We leave it to the reader to check that *GetSource*'s implementation satisfies its specification.

The loop that constructs the solution is called the *Solution Loop*. It is easily coded using *GetSource*, as shown in Figure 2.

For each $i$ from 0 to $N - 1$, the Solution Loop generates instruction $i$ of the solution program by setting the variable *tmpl* to the template of which instruction $i$ is an instance and setting the variable $\theta$ to the substitution of actuals for template parameters by which instruction $i$ instantiates *tmpl*. Having done this, the loop simply emits the instruction $\theta(\texttt{syntax}(tmpl))$.

$res := GetSource(i, j, t, c)$

**specification**:

**requires** $s \models L(i, t, c)$
Let $t$ be $f(q_0, \ldots, q_{m-1})$. Then $GetSource(i, j, t, c)$
**ensures** that the return value *res* satisfies these five conditions:
(0) *res* is a component or a numeral
(1) If arg $j$ to $f$ is source-restricted to $c$, then *res* is $c$.
(2) if *res* is a component, then $s \models H(i, q_j, res)$
(3) if *res* is a numeral, then $res \in q_j$
(4) if $src_j(f)$ is a template parameter,
then $s \models B(i, template(f), src_j(f), res)$

**implementation**:

**if** arg $j$ of $f$ is source-restricted to $c$ **then**
$res := c$
**elseif** arg $j$ of $f$ is the constant numeral $n$ **then**
$res := n$
**elseif** arg $j$ to $f$ is a free-source argument **then**
By the free-source case of the argument constraint,
some register $d$ in $dest(class(t))$ exists
such that $s \models H(i, q_j, d)$ **and** $B(i, template(f), src_j(f), d)$.
Pick any such $d$ and set $res := d$.
**elseif** arg $j$ to $f$ is an in-place argument **then**
set $res := c$
**elseif** arg $j$ to $f$ is an immediate argument **then**
$res :=$ the numeral in $q_j$
**end**

Fig. 1. The subroutine used by the solution loop to determine the source of a given argument of a given update of a given instruction of the solution.

Setting *tmpl* is easy: the loop sets it to $template(f)$ for the operation $f$ of any term $t$ for which $s \models L(i, t, c)$ for some $c$. For any such $t$, the first case of the issue constraint implies $s \models U(i, template(f))$, and the second case of the issue constraint implies that for any $i$, at most one unknown $U(i, tmpl)$ is true in $s$. Therefore, the different $c, t$ pairs enumerated for $i$ all lead to the same setting of *tmpl*. That is, all of the updates scheduled for instruction $i$ are applications of machine operations that have the same template.

To set $\theta$, the Solution Loop initializes $\theta$ to be the empty substitution and updates $\theta[\#k] := cj$ whenever instruction $i$ contains a template parameter `#k` which appears as argument $j$ of one of the terms of an update of the instruction, and *cj* is the source of the corresponding actual parameter. (Where "source" is defined as in the description of *GetSource*.) In this case, *GetSource* postcondition (4) implies that $s \models B(i, template(f), src_j(f), cj)$. And the third part of the issue constraint allows $s \models B(i, template(f), src_j(f), cj)$ to be true for only one *cj* for any particular triple of previous arguments to $B$. Consequently, the binding introduced to $\theta$ for an argument in one $c, t$ pair is not overwritten when working on a later argument or on a later $c, t$ pair. The Solution Loop also sets $\theta[\#k] := c$ if instruction $i$ must store into register $c$ the value of an application of a machine operation $f$, where

```
The Solution Loop:
  for i := 0 to N − 1 do
    var tmpl := nil, θ := the empty substitution in
      for each c, t such that s ⊨ L(i, t, c) do
        let t be f(q_0, ..., q_{m−1}) in
          tmpl := template(f);
          for j := 0 to m − 1 do
            var cj := GetSource(i, j, f(q_0, ..., q_{m−1}), c) in
              if src_j(f) is a template parameter then
                θ[src_j(f)] := cj
              end
            end
          end ;
          if target(f) is a template parameter then
            θ[target(f)] := c
          end
        end
      end;
      emit the instruction θ(the syntax of tmpl
    end
  end
```

Fig. 2. The loop that constructs a solution to the automatic programming problem from a solution $s$ of the satisfiability constraints.

*target*$(f)$ is a register parameter #k. Because of the result constraint and part three of the issue constraint, this update cannot interfere with other updates to $\theta$ for instruction $i$.

The code in Figure 2 omits two fine points. First, if after the loop on $c$, $t$, the template *tmpl* is **nil**, then no operations have been allocated to instruction $i$, and we can emit a no-op instruction. But this will never happen for the optimal budget, and it is only for the optimal budget that there is any need to transform the satisfiability solution into a code sequence. Second, if, after the loop on $c$ and $t$, the template *tmpl* is defined but some template parameter #k of *tmpl* remains unbound, then each such $\theta$[#k] must be set to some arbitrary legal actual before using $\theta$ to emit the assembly instruction. This situation could arise, for example, if the optimal code depends on some but not all of the updates of an instruction.

## 4. THE CORRECTNESS THEOREM

**Theorem.** For any $c, q, n$ such that $s \models H(n, q, c)$, the first $n$ instructions of the program generated by the Solution Loop terminate in a state where component $c$ holds the common value of all terms in $q$.

It is appropriate to call this the correctness theorem, since, combined with the Completion Constraint, it shows that the code generated by the Solution Loop implements the goal multi-assignment within the budget, that is, that it is correct.

We will prove the theorem by induction on $n$.

Base case, $n = 0$. For any $q, c$ such that $s \models H(0, q, c)$, we have by the initialization constraint that $c$ is an element of $q$. Hence the common value of the terms in $q$ is given by the term $c$.

Induction step, $n > 0$. Suppose the claim is true for all $q$, $c$, and for $n-1$. Let $q$ and $c$ be such that $s \models H(n, q, c)$. Consider the two cases of the hold constraint whose antecedant is $H(n, q, c)$.

The first and more interesting case is that $s \models L(n-1, t, c)$ for some $t$ in $q$, where $t$ is of the form $f(q_0, \ldots, q_{m-1})$. Then in constructing instruction $n-1$ the Solution Loop enumerated the pair $t, c$, and when working on this pair, the loop called *GetSource*$(n-1, j, t, c)$ for each $j$ from 0 to $m-1$. For each of these $j$'s, let $x_j$ be the value returned by *GetSource*. We argue that in the prestate of instruction $n-1$, each $x_j$ is a numeral denoting the value of $q_j$ or a component that contains the value of $q_j$. For those $j$ where $x_j$ is a numeral, this follows from *GetSource* postcondition (3). For those $j$ where $x_j$ is a component, this follows from postcondition (2) and induction. In either case it follows that $x_j$ denotes the value of $q_j$ in the prestate of instruction $n-1$. We also argue that the instantiated instruction contains an update that applies $f$ to $x_0, \ldots x_{m-1}$ and stores the result in $c$. For each $j$ such that $src_j(f)$ is a template parameter `#k`, the loop sets $\theta$[`#k`] to $x_j$ and then later uses $\theta$ to instantiate the template, so the result will be to produce an instruction that applies $f$ to an argument vector whose element $j$ is $x_j$. For any $j$ such that $src_j(f)$ isn't a template parameter, it must be a fixed component, and *GetSource*$(i, j, t, c)$ will return that component (according to *GetSource* postcondition (1)), hence in this case the instantiated instruction also has $x_j$ as argument $j$ to $f$. So the update in instruction $n-1$ is equivalent to $c := f(x_0, \ldots, x_{m-1})$, and, since each $x_j$ has the value of $q_j$ and the term $t$ is $f(q_0, \ldots, q_{m-1})$, this sets $c$ to the value of $t$. So in the post-state of instruction $n-1$, $c$ contains the value of $t$, which, since $t$ is an element of $q$, is the common value of all terms in $q$.

The second case of the hold constraint is that (1) $s \models H(n-1, q, c)$ and (2) for no $u$ does $s \models L(n-1, u, c)$. By (1) and induction, we see that after the first $n-1$ instructions, $c$ holds the value of $q$. Examining the Solution Loop with (2) in mind shows that the $n$th instruction (instruction $n-1$) doesn't modify $c$. Therefore $c$ still holds $q$'s value after instruction $n-1$. These two cases complete the proof of the inductive step, which completes the proof of the correctness theorem.

The correctness theorem (combined with the Completion constraint) shows that the satisfiability constraints are strong enough, that is, that the program generated from the satisfiability constraints is correct. To prove that the search over various budgets produces a program that is optimal as well as correct, we must also prove that the satisfiability constraints are weak enough, that is, that they are unsatisfiable only when the automatic programming problem from which they were constructed has no solution. We believe that we know how to prove this, but the only part of the proof we have written down is the proof of the Destination Lemma.

## 5. CONCLUDING REMARKS

We have presented an algorithm for a super-optimizer that uses a simple declarative architectural description mechanism. It is much slower than conventional optimizers, but it really is an optimizer, not just a code improver. And we have evidence that it will be fast enough to be of engineering utility.

It remains to be seen whether we can extend the Denali-2 algorithm to minimize the number of cycles instead of just the number of instructions. Denali-1 did achieve this in the case of the EV-6 superscalar architecture. This required us to include both instruction indexes and cycle indexes among the unknowns. It was more complicated and less tabledriven than the algorithm presented in this paper. For the EPIC architecture IA-64, the situation is different from the super-scalar case: we need to construct a program that would be correct if it were executed one instruction at a time, but that also meets the EPIC restrictions that allow groups of adjacent instructions to multi-issue. It seems, therefore, that we will be able to reuse the initialization constraint, the argument constraints, the result constraint, the issue constraints, and the two-case hold constraint, and that only the completion constraint is irrelevant to the cycle-minimizing setting. Instead of the completion constraint, we will have to choose the instructions and a partition of the instructions into groups in such a way as to minimize the number of groups that contain an instruction that reads a register value written by a previous group, which is the fundamental limitation to multiple issue in EPIC architectures. This doesn't seem too daunting when put in so few words. But (1) any actual implementation of IA-64 has additional limits on multiple issue besides the fundamental one and (2) the devil is in the details, and we know we will face many details before we have a working super-optimizer for the Madison implementation of IA-64.

Not only does the Denali method produce code that is superior to that produced by "optimizers" that are actually code improvers, but also the Denali method is rather simple, at least compared to serious optimizing code generators. Of course this paper has omitted many details, but everything really fundamental to the method is explained in the 28 combined pages of the Denali-1 paper and the present paper, or in the 31-page description of E-graph matching in the comprehensive paper about the Simplify prover [1]. It is a liberating simplification to relegate all the backtracking search issues to the satisfiability solver, where they are part of someone else's code, not ours. We also feel that our approach produces an algorithm that is pleasantly declarative, since a large part of the program is input to the satisfiability solver and matcher. The Denali-1 prototype is less than fifteen thousand lines of Java.

Admittedly this accounting exaggerates Denali's simplicity, since, by the time we implement software pipelining and other conventional optimizations that we haven't yet included, the source code will doubtlessly grow and no longer be impressively small.

But it is nice to start with a simple foundation.

## REFERENCES

David L. Detlefs and Greg Nelson and James B. Saxe. Simplify: A theorem-prover for program checking. Research Report HPL-2003-148, HP Laboratories Technical Report, Palo Alto, USA, jul 2003.

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *JACM*, 27(4):758–771, October 1980.

Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM 2000 Conference on Programming Language Design and Implementation*, pages 304–314, june 2002. Berlin.

Henry Massalin. Superoptimizer: a look at the shortest program. *Proceedings of the second international conference on architectural support for programming languages and operating systems*, pages 122–26, 1987.