Optimization of Functional Programs by Grammar Thinning

ADAM WEBBER
Western Illinois University

We describe a new technique for optimizing first-order functional programs. Programs are represented as graph grammars, and optimization proceeds by counterexample: when a graph generated by the grammar is found to contain an unnecessary computation, the optimizer attempts to reformulate the grammar so that it never again generates any graph that contains that counterexample. This kind of program reformulation corresponds to an interesting problem on context-free grammars. Our reformulation technique is derived from an (approximate) solution to this CFG problem. An optimizer called Thinner is the proof of concept for this technique. Thinner is a fully automatic, source-to-source optimizer for a Lisp-like language of purely functional, first-order programs. Thinner rediscovers a wide variety of common compiler optimizations. It also finds other more exotic transformations, including the well-known Fibonacci reformulation and the Knuth-Morris-Pratt optimization.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars—*grammar types*

General Terms: Languages, Theory

Additional Key Words and Phrases: Functional languages, graph grammars, optimization

1. INTRODUCTION

Both of the following Lisp functions stand in need of optimization:

```
(defun fib (a)
   (if (< a 2)
      a
      (+ (fib (- a 1))
         (fib (- a 2)))))

(defun cse (a)
   (f (* a 10) (* a 10)))
```

In both cases, the clear optimization involves the elimination of redundant computation. Yet the first is considered difficult while the second is trivial. This is because the recursive structure of the first program conceals the redundant com-

This research was supported in part by NSF grant IRI-8902721.
Author's address: Department of Computer Science, Western Illinois University, Macomb, IL 61455; email: ab-webber@bgu.edu.

putation and obscures the transformation that would get rid of it. But while this camouflage strains the intellect of human programmers, it can be penetrated by an automatic optimizer.

In this article we describe a technique called *grammar thinning* that handles the elimination of redundant computation from programs. Grammar thinning is a technique that belongs with partial evaluation and supercompilation, as a powerful (and expensive) tool for general optimization and deep program transformation.

We begin with an introductory example and a discussion of related work. Section 2 treats in detail the grammar-thinning problem for context-free grammars. To use grammar thinning for program optimization, programs must be represented grammatically. Section 3 develops the *trace grammar* representation and gives a rigorous definition of the kind of redundancy-eliminating optimization our method attempts. Section 4 describes Thinner, a fully automatic, source-to-source optimizer for a Lisp-like language of purely functional, first-order programs. Using grammar thinning, Thinner discovers a variety of commmon compiler optimizations (common-subexpression elimination, constant folding, dead-code elimination, code sinking, loop invariant removal, loop jamming, and loop splitting), along with other more difficult reformulations. Section 5 offers some conclusions and open problems.

1.1 Example

To give an overview of the method and terminology we begin by sketching an example: the Fibonacci optimization. Thinner begins by compiling the Fibonacci function shown above into a trace grammar, shown at the top of Figure 1. Thinner's inference engine reasons about graphs derivable in the grammar, starting with those that actually occur as right-hand sides of productions and moving on to those that can be obtained by one or more additional steps of derivation. This blind search eventually locates a graph that contains an unnecessary computation.

The *thinning example* δ, shown in Figure 1, is extracted from that graph. A thinning example is a subgraph that isolates a redundant computation. Thinner also constructs a *kernel set* K: these are minimal subgraphs that could expand into graphs that contain the thinning example. The concept of the kernel is a very important one for the method. The goal of the method is to optimize all instances of the thinning example that can be derived. To do this it alters not only the thinning example itself, but all nontrivial precursors of that example in any derivation. For the given grammar and δ there is a single kernel. Note that when one of the kernel graph's nonterminals (the one that computes `(fib (- n 1))`) is expanded, the resulting graph contains the thinning example.

To solve the thinning problem shown in Figure 1, we first "thin" the right-hand sides of the productions for `fib` (that is, eliminate any unnecessary code) and then examine the resulting graphs for kernels. Thinning the right-hand sides has no effect (since the thinning example does not occur yet). But there is an instance of our kernel in the second production. The productions for `fib` after this step are shown in Figure 2; kernel vertices have been grouped together.

The next step is to replace the kernel-matched group shown in Figure 2 with a new nonterminal X_β. This yields the productions shown in Figure 3.

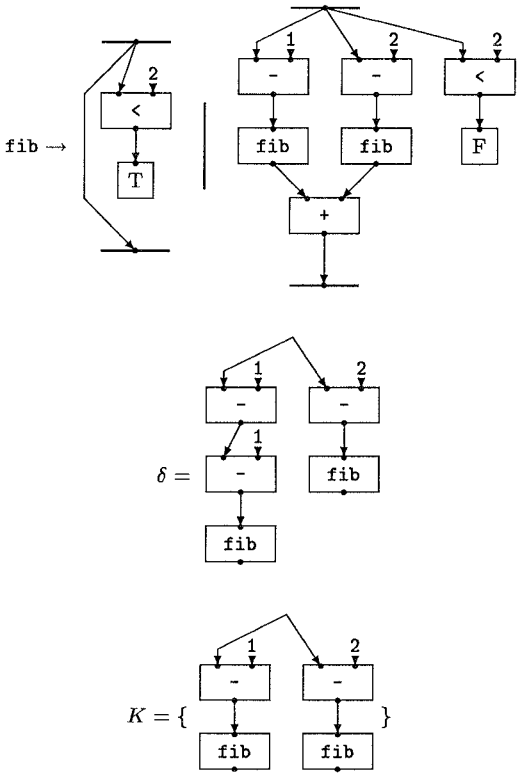


Fig. 1. A trace grammar thinning problem.

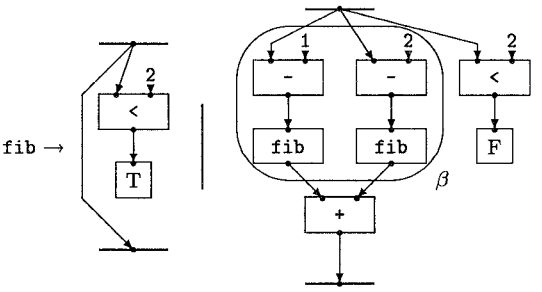


Fig. 2. Productions for `fib` after thinning and grouping.

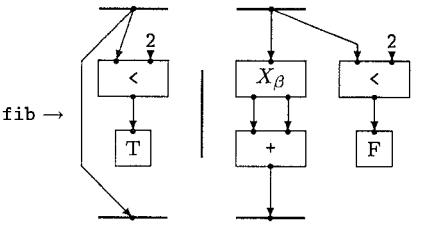


Fig 3. Productions for `fib` after replacement of the kernel group.

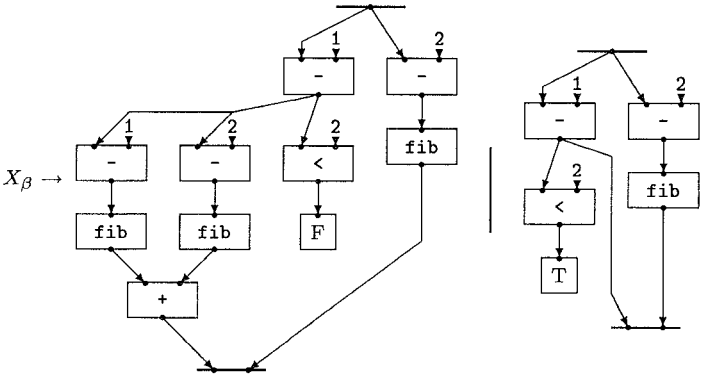


Fig. 4. Productions for X_β.

Next, we choose a nonterminal to expand, and we expand it. There are two `fib` nonterminals in the group β, but the one that computes `(fib (- n 1))` is the one whose expansion yields an instance of the thinning example. We unfold that nonterminal: there are two productions for `fib` in the original grammar, so this unfolding yields two possible right-hand sides for X_β, as shown in Figure 4.

Now the graphs shown in Figure 4 are thinned and grouped. Thinning does have an effect this time, since the first graph contains an instance of the thinning example. After thinning, the group β is again found to occur in that graph. The second graph shown in Figure 4 is carried into the new grammar unchanged, and the resulting productions for X_β are shown in Figure 5.

Finally, we replace the group β with the nonterminal X_β. The final grammar is shown in Figure 6. It is an exact solution: a grammar that computes the same function as the original, but never generates a graph that contains the thinning example. The grammar shown in Figure 6 is not yet as thin as possible: using a different thinning example, the use of `fib` in the second production for X_β can be eliminated. But the grammar is significantly thinner than the original. In particular, it corresponds to a linear-time, rather than an exponential-time, function.[1]

[1]Actually, calling this linear time is a bit of a stretch since the size of the output is exponential in the size of the input. But most authors make the same stretch [Rohl 1989].

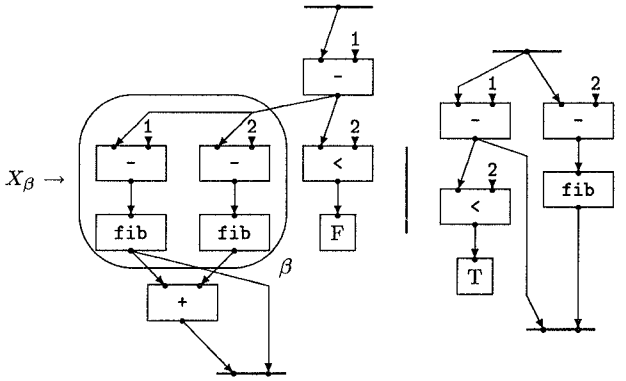


Fig. 5. Productions for X_β after thinning and grouping.

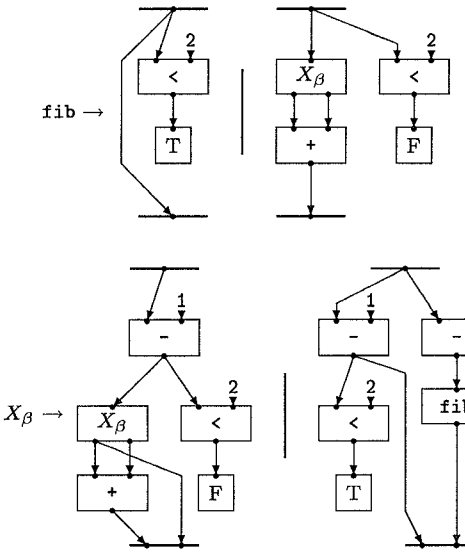


Fig. 6. Final grammar.

1.2 Related Work

1.2.1 *Irrelevance Reformulation.* The Thinner project is a descendant of Subramanian's work on irrelevance in first-order theories [Subramanian 1989; Subramanian and Genesereth 1987]. Subramanian characterized a formal principle corresponding to the intuitive idea of irrelevance. This formal principle served as justification for the automatic reformulation of first-order theories. Subramanian's thesis mentions the possibility of applying an irrelevance principle to the problem of eliminating repeated computation in a logic-programming formulation of the Fibonacci function.

1.2.2 *Hall's Optimizer.* The Thinner project is related to the optimizer described in Hall's [1990] thesis. Hall's optimizer achieves considerable power by making use of sources of information other than the raw program: for example, user-supplied test inputs, optimization invariants, and proofs of correctness, as well as syntactic

clues like iteration macros for series expressions. Another source of power is the fact that optimizations are not guaranteed to be correct (except with respect to the given test inputs). We have taken a more circumscribed approach in the Thinner project. Our only input is the raw, purely functional program, and the optimizations produced by the Thinner are guaranteed correct. Within this circumscribed domain we show a clear derivation of optimizations from a rigorous statement of an optimizing principle, using a consistent graph-grammar treatment of recursive program structure.

Hall's optimizer uses several transformations: the redistribution of intermediate values and removal of disconnected functions, the elimination of unnecessary copy operators (when destructive operators are used), the combination of series expressions [Steele 1990] in "generalized loop fusion," the addition of new parameters to a function, and the addition of new returned values from a function. Hall's system does not perform deeper grammatical transformations like the Fibonacci reformulation (although he suggests ways to extend his optimizer for that particular example). Thinner uses grammar thinning, which in many cases (including the Fibonacci case) eliminates all instances of unnecessary computation covered by a given example.

1.2.3 *Redundancy Elimination.* The term "redundancy elimination" occurs in the literature surrounding a particular kind of transformation for imperative programs. This transformation is a generalization of common-subexpression elimination with code motion.

Value numbering [Cocke and Schwartz 1970] is the parent of more-modern methods [Downey et al. 1980] of identifying redundant expressions. A computation is fully redundant if there is an equivalent computation before it on every path of control flow that reaches it. Computations may also be partially redundant: redundant on some but not all paths. Morel and Renvoise [1979] addressed the problem of eliminating partial redundancies, and later work built on this idea [Rosen et al. 1988].

This kind of redundancy elimination is only loosely related to the Thinner project. Many of the problems of the area are unique to imperative programs.

1.2.4 *Advanced Functional Optimization.* The algebraic approach developed in Backus [1985] is another advanced method of optimization for functional programs. The foundation of this approach is an axiomatic semantics for the language FP. From these axioms Backus proved several theorems about identities: one of these is a "recursion removal" theorem which justifies a class of transformations of recursive functions to iterative form. Kieburtz and Shultis [1981] used a similar approach and developed additional theorems. The mathematical elegance of the algebraic approach is attractive, but for our purposes the question of whether automatic transformations are proved correct by this or by some other formal means is moot.

Another relevant method of optimization is the patterns-of-redundancy approach advanced by Cohen [1983]. He gave a taxonomy of several types of redundancy: explicit, common-generator, commutative, and so forth. In Cohen's approach, a program can be classified in one of these categories by fitting it into a fixed recursion schema and identifying properties (like commutativity) of its primitive functions. Cohen identified several important coarse-grained patterns of unnecessary

computation with this approach. He did not attempt to give a flexible collection of primitive transformations, and he did not address the problem of automation. Indermark and Klaeren [1987] took a similar approach: they were content to identify a specific high-level pattern of redundant computation (Fibonacci-like recursion).

1.2.5 *The Transformational Method.* The transformational method was pioneered by Burstall and Darlington [1977]. They presented a small set of transformations including the *fold* and *unfold* operations for systems of recursion equations. This set of transformations is sufficiently fine grained that a wide range of program transformations can be composed from it. The missing piece in this approach is a fully automatic way of deciding when to apply the transformations.

Partial evaluation [Consel and Danvy 1993] and supercompilation [Turchin 1980; 1986; Turchin et al. 1982] can be seen as fold/unfold strategies for this transformational method. A comparison of these and other fold/unfold strategies was undertaken by Sørensen et al. [1994]. They compared the fold/unfold methods on their ability to perform the KMP optimization—the optimization of a general pattern matcher for a particular fixed pattern, as performed by the Knuth-Morris-Pratt algorithm [Knuth et al. 1977].

The method of kernels used by Thinner is another fold/unfold strategy. Kernels are derived automatically from the thinning example (which is derived automatically from the program). In effect, Thinner decides when to fold and unfold by pursuing the goal of exposing instances of a particular thinning example. Kernels correspond roughly to the "eureka definitions" commonly reported in the literature on the transformational method, in the sense that they are used to guide folding. But there is no one-to-one relation between kernels and new function definitions. Any subgraph that matches a kernel must be folded together, and any piece that is folded will yield a new function definition. But if several kernels overlap, the new function definition will not match any single kernel; and if a kernel never matches a subgraph without overlapping (or never matches at all), no function definition will be generated for it.

Thinner does perform the KMP optimization. This has less to do with the fold/unfold strategy than with the amount of information available to the optimizer at each point in the process. Thinner applies its inference technique to the full trace graph, once the thinning example is exposed; in the terminology of Sørensen et al. [1994], it has both positive and negative information. But from a grammatical point of view, the KMP optimization is a relatively simple one—the "mistake" the unoptimized pattern matcher repeats (performing comparisons with constants) always fits within the existing recursive structure of the program. Examples that test a transformational method more strenuously are the Fibonacci reformulation and the functional equivalent of loop invariant removal, because in these examples the mistake the unoptimized program repeats requires repairs that cross recursive function invocations.

2. THE CFG-THINNING PROBLEM

As mentioned above, our method of optimization works by counterexample. The optimizer discovers an inefficient path through the program, then tries to reformulate the program to eliminate *all* instances of that same inefficiency. We call this

kind of reformulation *grammar thinning*. There is an interesting, slightly simpler grammar-thinning problem for context-free grammars on strings; an (approximate) solution to the CFG problem is the basis of our optimization technique.

2.1 Fully Terminal CFG Thinning

In the context-free grammar version of the thinning problem we are given a CFG and a thinning example which is a string δ with one or more marked characters. The example says, in effect, "The marked symbols in this sequence are unnecessary," and the problem is to modify the CFG to incorporate the lesson of this example. What does this mean precisely? When δ is a fully terminal string the problem is fairly easy to state rigorously, and we will deal with this case first.

Suppose the thinning example δ contains terminal symbols only. We define a function ThinString for editing a terminal string x using a thinning example δ as follows.

```
function ThinString(x, δ);
    x is a string of terminals;
    δ is a string with at least one marked symbol;
    ThinString(x, δ) is a string of terminals (a subsequence of x);
begin
    while there is an instance[2] of δ in x
        begin
            find the leftmost instance of δ in x;
            delete from x those symbols matched to marked symbols in δ
        end;
    return x
end;
```

For example, if $x = aabc$ and $\delta = \underline{a}b$ (where underlining indicates a marked character), ThinString deletes symbols from x to end up with the string bc: first it finds the instance of $\underline{a}b$ in $aabc$ and deletes that a to get abc; then it finds the instance of $\underline{a}b$ in abc and deletes that a to get bc. The reader may wish to try an example or two. Try thinning the string $xyxxyxxyxy$ using $\delta = \underline{x}yxy$. (Remember to thin the *leftmost* instance of δ each time.) Or try thinning $abxaabxabbxaabxabbxab$ using $\delta = \underline{a}b\underline{x}a\underline{b}$. In both examples the thinning step is repeated five times.

Now let G be a context-free grammar, and consider the language $L_\delta(G)$ obtained by thinning each word in $L(G)$ using the string δ, i.e.,

$$L_\delta(G) = \{w \mid w = \text{ThinString}(v, \delta) \text{ for some } v \in L(G)\}.$$

This is like learning the lesson of δ ex post facto: first make all the mistakes of $L(G)$; then go back and correct them. What we want, of course, is a new CFG H for which $L(H) = L_\delta(G)$. It turns out that this is not always possible.

THEOREM 2.1.1. *There exist a CFG G and a thinning example δ for which $L_\delta(G)$ is not context free.*

PROOF. One can construct the language $\{a^n b^n c^n \mid n \geq 0\}$ using only operations that are closed for the CFLs, plus thinning. See the proof of Theorem 2.4 in Jantzen

[2]By "instance" we mean substring or, as some authors have it, factor.

$M(\delta)$ regular	$M(\delta)$ not regular
$\underline{a}\underline{b}$	$\underline{a}\underline{b}$
$\underline{x}yx\underline{y}$	$x\underline{x}z\underline{y}\underline{y}$
$\underline{a}\underline{a}\underline{a}$	
$\underline{a}\underline{b}\underline{x}\underline{a}\underline{b}$	

Fig. 7. Some thinning examples classified.

and Petersen [1994].[3] □

However, there is a special case (actually, a common case) for which the thinning problem can be solved. This case allows a general CFG G but restricts the thinning example δ as described below.

Define a language $M(\delta)$ of strings "correctly marked for thinning" as follows: a string z is in $M(\delta)$ if and only if exactly those characters deleted when z is editing by δ are marked in z. So $M(\underline{a}b)$ includes $\underline{a}bc$ and $\underline{a}\underline{a}bc$ but not abc (because a character deleted during editing by $\underline{a}b$ is not marked) and not $\underline{a}c$ (because a marked character is not deleted). The language $M(\delta)$ appears to be context free for all δ, but this is unproven.

CONJECTURE 2.1.2. *For any δ, $M(\delta)$ is a context-free language.*

It often happens that $M(\delta)$ is not only context free but regular. This is the kind of δ for which we have a solution to the CFG thinning problem.

THEOREM 2.1.3. *If $M(\delta)$ is a regular language then $L_\delta(G)$ is context free for any context-free grammar G.*

PROOF. Suppose G is a CFG over an alphabet Σ, and δ is a thinning example for which $M(\delta)$ is a regular language. If L is a language over Σ without any marked symbols, define $S_\Sigma(L)$ to be the same language but allowing all patterns of marked and unmarked symbols. That is, S_Σ is a substitution applied to L, which maps each terminal symbol $x \in \Sigma$ to $\{x, \underline{x}\}$. Clearly $S_\Sigma(L(G))$ is context free. Since $M(\delta)$ is regular, the intersection of $S_\Sigma(L(G))$ and $M(\delta)$ is context free. This is the language $L(G)$ but with each string correctly marked for thinning; so by substituting the empty string for each marked symbol, we arrive at $L_\delta(G)$ and conclude that it is context free. □

This proof suggests a method for transforming grammars—a method we have implemented. The tricky part is the construction of a CFG for the intersection of two languages, one a context-free language given by a CFG and the other a regular language given by an FSM. Salomaa [1973] gives a treatment of this problem.

Since we have a solution to the fully terminal thinning problem when the thinning example δ yields a regular $M(\delta)$, it makes sense to ask whether there is a more direct characterization: for what kinds of δ is $M(\delta)$ a regular language? Figure 7 shows a selection of thinning examples δ, categorized according to whether $M(\delta)$ is regular. Note that $M(\delta)$ is regular for many thinning examples: it is regular whenever δ has exactly one marked symbol, and it is regular even for some fairly tricky δ like $\delta = \underline{x}yx\underline{y}$ and $\delta = \underline{a}b\underline{x}\underline{a}\underline{b}$ (which were used above in an exercise for the reader).

All thinning examples of which we are aware fit a simple pattern:

[3] Thanks to J. Engelfriet and H. J. Hoogeboom for pointing out this proof.

CONJECTURE 2.1.4. *If $M(\delta)$ is nonregular then thinning δ yields some string β for which there are nonempty strings α and γ such that $\delta = \alpha\beta\gamma$ (and so $\alpha^n\beta\gamma^n$ thins to β).*

The implication does not work the other way, as shown by the case $\delta = \underline{a}a\underline{a}$. Perhaps a strengthened form of this condition will prove to be both necessary and sufficient.

2.2 General CFG Thinning

The trouble with the terminal solution presented above is that it will not yield a particularly general optimization method for programs. Consider the exponential-time Fibonacci function, or loop invariant removal, or loop jamming: these are transformations in which the thinning example will definitely contain function calls (that is, nonterminal vertices). We could extend the terminal solution to make it treat some nonterminals as terminals, but this would only work if the thinned nonterminals never participate in an instance of δ. The Fibonacci example does not appear to fit this case: the thinned nonterminal is a recursive call. We need a different approach.

Suppose δ contains nonterminals. How can we edit a language to reflect the lesson of this δ? It seems natural to extend the original string-thinning function to parse trees. Define the *yield* of a tree to be the string of its leaves, read from left to right; and define a *prefix* of a tree to be that tree with any number (zero or more) of its nonterminals left unexpanded. Then we can define an *instance* of δ in a tree T to be a string of nodes of T that matches δ and is a substring of the yield of a prefix of T.

```
function ThinTree(T, δ);
    T is a parse tree;
    δ is a string with at least one marked symbol;
    ThinTree(T, δ) is a tree, a pruned version of T;
begin
    while there is an instance of δ in T
        begin
            find the lexically first instance of δ in T;
            delete from T those subtrees matched to marked symbols in δ
        end;
    return T
end;
```

This is an extension of ThinString in the sense that if δ is fully terminal, it makes no difference whether you thin the yield of a parse tree using ThinString or whether you thin the parse tree using ThinTree and then take the yield: the result is the same. Proceeding as before, we can define the language $L_\delta(G)$ to be the language obtained by editing each parse tree generated by G using the string δ, then reading off the remaining terminal string, i.e.,

$$L_\delta(G) = \{w \mid w \text{ is the yield of ThinTree}(T, \delta) \text{ for some } T \text{ generated by } G\}.$$

Again, when δ is fully terminal, this is the same as the previous definition of $L_\delta(G)$. But we now have a meaningful definition even when δ contains nonterminal symbols.

It may be a meaningful definition, but it seems to lead to an intractable problem! We made progress in the fully terminal case by insisting that $M(\delta)$ be regular; in the nonterminal case, since we are not editing strings, the question of whether the string language $M(\delta)$ is regular seems irrelevant.

2.3 The Method of Kernels

Since our ultimate motivation is the optimization of trace grammars, we need not despair just because there is no perfect solution to the thinning problem. An approximate one, one that eliminates some but not necessarily all instances of the thinning example, will do.

Consider the thinning example $\delta = b\underline{Sb}$: how can this arise as a substring of some sentential form in the grammar $S \rightarrow SbSc \mid a$? It can arise only from a sentential form $\alpha bS\beta$ (for any strings α and β) by the expansion of the S using production $S \rightarrow SbSc$. The string bS is therefore a *kernel* of δ in the grammar: it is a minimal string of symbols that expands into a string containing δ. Now, how can bS arise? It can only arise from the nonterminal S, when that nonterminal is expanded using the production $S \rightarrow SbSc$.[4]

More formally, we define a kernel as follows:

Definition 2.3.1. A *kernel* of a thinning example δ in a grammar G is a string ϕ which is not a single nonterminal and not δ and which has the property that $\phi \Rightarrow_G^* \alpha\delta\beta$ for some strings α and β, in such a way that every symbol of ϕ derives a part of δ.

If we remove ϵ-productions from the grammar, we can guarantee that for every kernel ϕ, $|\phi| \leq |\delta|$ (and it follows that there are finitely many kernels).

Our plan will be to attempt to prevent the occurrence of kernels by modifying the grammar so that each kernel is replaced by a new nonterminal. The following procedures, ThinByKernels and Kernelize, summarize this method.

```
function ThinByKernels(G, δ, K);
    G is a context-free grammar;
    δ is a thinning example;
    K is a set of kernels for δ in G;
begin
1   G' := a new, empty grammar;
2   for each production A → α in G do
3       Kernelize(A, α, G, G', δ, K);
4   return G'
end;

procedure Kernelize(A, α, G, G', δ, K);
    A is a nonterminal;
    α is a string of symbols;
    G is a context-free grammar;
    G' is a context-free grammar, which we augment;
    δ is a thinning example;
```

[4] This is not quite true: it can also arise from itself. But since we are only interested in enumerating the distinct kernels of δ in G this is irrelevant.

```
    K is a set of kernels for δ in G;
begin
1   α := ThinString(α, δ);
2   for each kernel k ∈ K do
3       for each instance of k in α do
4           note that α cannot safely be broken within this k;
5   break α into groups wherever it can safely be broken;
6   break α further if necessary to keep group size below some constant c;
7   for each group β in α with |β| ≥ 2 do
8       begin
9           replace β in α with a new nonterminal X_β;
10          if X_β is not yet a nonterminal in G' then
11              begin
12                  choose a nonterminal in β;
13                  for each β' derived by expanding that nonterminal using G do
14                      Kernelize(X_β, β', G, G', δ, K)
15              end
16      end;
17  add production A → α to grammar G'
end;
```

The Kernelize procedure is not fully specified: line 6 does not explain how groups are to be broken up to keep the group size below c, nor does it say what c should be; and the procedure by which a nonterminal to expand is chosen at line 12 is not specified. We will return to these points shortly, but first let us walk through a simple example, assuming for now that c is large enough that we always have $|\beta| < c$ at line 6. Let G be the grammar $S \rightarrow SbSc \mid a$, and let δ be $b\underline{Sb}$. As noted previously, the only kernel for δ is the string bS; so let $K = \{bS\}$. What happens when we ThinByKernels(G, δ, K)?

There are two productions in G: $S \rightarrow a$ and $S \rightarrow SbSc$. They yield two different calls to Kernelize at line 3 in ThinByKernels. Kernelize(S, a, G, G', δ, K) simply adds the production $S \rightarrow a$ to G'.

Kernelize$(S, SbSc, G, G', \delta, K)$ is more complex. The thinning at line 1 has no effect, since δ does not occur in $\alpha = SbSc$. At lines 2-4 we observe that there is an instance of the kernel bS in α, so the string cannot be broken between those symbols. Thus, at line 5, we break $\alpha = SbSc$ into $S \cdot bS \cdot c$; and the only group β on which lines 7-16 are executed is $\beta = bS$. We change α from $SbSc$ to $SX_{bS}c$ at line 9. The nonterminal X_{bS} is not yet present in G', so we execute the loop at lines 13-14: there is only one nonterminal in β that can be expanded, and this expansion yields two values for β': ba and $bSbSc$. Using these two values of β', we use Kernelize recursively. Kernelize$(X_{bS}, ba, G, G', \delta, K)$ simply adds the production $X_{bS} \rightarrow ba$ to G'.

Kernelize$(X_{bS}, bSbSc, G, G', \delta, K)$ is more complex. First of all, the string $\alpha = bSbSc$ does contain an instance of the thinning example, and it is thinned to $\alpha = bSc$ at line 1. Now it contains an instance of the kernel bS, so it is broken into groups as $bS \cdot c$, and the bS is replaced by X_{bS}. This nonterminal is already part of G', so lines 13-14 are not executed. The procedure adds the production $X_{bS} \rightarrow X_{bS}c$ to G', and terminates. The resulting grammar G' has productions $S \rightarrow SX_{bS}c \mid a$ and $X_{bS} \rightarrow X_{bS}c \mid ba$. This is an exact solution: a grammar G' such that $L(G') =$

$L_\delta(G)$.

2.4 Shortcomings of the Method of Kernels

Why is the grammar returned by ThinByKernels not always an exact solution? There are really two sources of inaccuracy. The first arises because we may break up some groups at line 6 (those of size c or greater). Without this step we cannot guarantee that Kernelize will terminate. Since kernels may overlap in lines 2-4, there is no limit to the length of the group β in line 5; so without the upper bound, there would be no limit to the number of different nonterminals X_β we might try to add to G'. Line 6 skirts this problem by imposing an arbitrary limit c on the size of groups. This results in guaranteed termination, but at the cost of potential incompleteness: by ignoring large groups Kernelize may overlook instances of the thinning example.

For example, suppose we have this grammar G:

$$S \to SS \mid a \mid b$$

and suppose we want to thin it with respect to the thinning example $\delta = a\underline{b}$. Note that SS and aS are kernels for δ in G; since these can overlap, any string in aS^* will be grouped together. Using the bound $c = 3$, ThinByKernels might construct the following grammar:

$$\begin{aligned} S &\to X_{SS} \mid a \mid b \\ X_{SS} &\to SX_{SS} \mid X_{aS} \mid bS \\ X_{aS} &\to aX_{SS} \mid aa \mid a \end{aligned}$$

This is thinner than the original grammar, but it is not an exact solution. (An exact solution can be obtained in this case using the FSM method of Section 2.1.) Without the group size bound ThinByKernels would diverge, trying to construct an infinite grammar like this:

$$\begin{aligned} S &\to X_{SS} \mid a \mid b \\ X_{SS} &\to X_{SSS} \mid X_{aS} \mid bS \\ X_{SSS} &\to X_{SSSS} \mid X_{aSS} \mid bX_{SS} \\ X_{SSSS} &\to \ldots \end{aligned}$$

A second source of inaccuracy lies in the choice at line 12 in Kernelize of a nonterminal in β to expand. Suppose there is more than one nonterminal in β; which one should be expanded? Suppose, for example, we are faced with the following grammar G:

$$\begin{aligned} S &\to AB \\ A &\to aAb \mid x \\ B &\to bBa \mid y \end{aligned}$$

And suppose we want to thin it with respect to the thinning example $\delta = \underline{A}b\underline{B}$. A thinned grammar is not difficult to construct: except for the tree that derives xy, all parse trees in G are thinned exactly once. Here is a grammar that generates $L_\delta(G)$:

$$\begin{aligned} S &\to xy \mid aby \mid bBa \\ B &\to bBa \mid y \end{aligned}$$

But the method of kernels fails on this problem. Clearly AB is a kernel for δ in G: it is a kernel in two different ways, since we get a string that contains δ either by expanding the A to aAb or by expanding the B to bBa. But therein lies the problem: when Kernelize encounters the group $\beta = AB$ at line 12, it must choose a nonterminal to expand. Expanding the A would yield this thinned grammar:

$$S \rightarrow X_{AB}$$
$$X_{AB} \rightarrow abB \mid xB$$
$$B \rightarrow bBa \mid y$$

which although thinner than the original, does not generate $L_\delta(G)$. Expanding the second would yield this thinned grammar:

$$S \rightarrow X_{AB}$$
$$X_{AB} \rightarrow bBa \mid Ay$$
$$A \rightarrow aAb \mid x$$
$$B \rightarrow bBa \mid y$$

which is not as thin as possible either. The problem is that we cannot expand either nonterminal in AB without potentially passing by an instance of δ: in some cases δ arises when A is not expanded, and in some cases δ arises when B is not expanded. As long as there is at least one nonterminal in β that is expanded on all derivations from β to a string containing δ, the problem can be avoided.

2.5 The Connection with Trace Grammars

For all its limitations, the method of kernels is the most general solution we have for the CFG thinning problem. As we will see, it carries over reasonably well to the trace grammar thinning problem, and it is the method used by Thinner. The problems of implementing the method for trace grammars are discussed in Section 4; for now, we present a CFG example that exactly parallels the trace grammar example presented in the introduction.

We will start with this CFG G:

$$S \rightarrow SbSc \mid a$$

And we will thin it with respect to $\delta = cb\underline{S}$ using the method of kernels. The set K of kernels is just $\{SbS\}$. For the sake of exposition, we will divide the operation of ThinByKernels into the same five steps presented for the optimization of the exponential-time Fibonacci function, shown in Section 1.1.

(1) Thin and group—in this step, the right-hand side of $S \rightarrow SbSc$ is thinned (which has no effect) and grouped into kernels, leaving $S \rightarrow SbS \cdot c$. ($S \rightarrow a$, the other production for S, is carried into the new grammar unchanged.)

(2) Replace—now the group SbS is replaced by a new nonterminal X_{SbS}, which yields $S \rightarrow X_{SbS}ca$.

(3) Choose and expand—next, we choose a nonterminal in $\beta = SbS$ to expand. In this case, it will be the first S, since the other is not expanded in the derivation from β to δ. We expand it using the original productions, which gives two possible right-hand sides for X_{SbS}: $SbScbS$ and abS.

(4) Thin and group—in this step, the right-hand side of $X_{SbS} \rightarrow SbScbS$ is thinned (which reduces it to $SbScb$) and grouped into kernels, leaving $S \rightarrow SbS \cdot cb$. ($X_{SbS} \rightarrow abS$, the other production for X_{SbS}, is carried into the new grammar unchanged.)

(5) Replace—finally, the group SbS is replaced by the nonterminal X_{SbS}, leaving $X_{SbS} \rightarrow X_{SbS}cb$.

The resulting grammar is:

$$S \rightarrow X_{SbS}c \mid a$$
$$X_{SbS} \rightarrow X_{SbS}cb \mid abS$$

which gives an exact solution for $L_\delta(G)$.

3. TRACE GRAMMARS

To use the method of kernels for program optimization we must represent programs grammatically. This section presents the trace grammar formalism, a representation that supports grammatical reformulation. Trace grammars also admit a simple formal statement of our guiding principle of optimization: the principle that programs should not do anything unnecessary. We call this the Principle of Least Computation or PLC.[5]

3.1 Trace Graphs

A program foreshadows a class of potential paths of execution, only one of which is realized for any given input. These individual paths of execution are simply graphs of the flow of data, mapping inputs to outputs by way of intermediate values. These values are taken to be elements of some set $\mathcal{U}$; the actual universe of values is not relevant at this point, except that it must include distinguished boolean values "true" and "false."

3.1.1 *Trace Graphs*

Definition 3.1.1.1. A *trace graph* G is a tuple $(V, E, \text{ins}, \text{outs}, \text{label})$. V is the vertex set. The function $\text{ins} : V \rightarrow \mathcal{N}$ assigns an input arity to each vertex; for each i, $1 \leq i \leq \text{ins}(v)$, we denote the ith input to vertex v as v_i^{in}. Similarly, the function $\text{outs} : V \rightarrow \mathcal{N}$ assigns an output arity to each vertex; for each j, $1 \leq j \leq \text{outs}(v)$, we denote the jth output of vertex v as v_i^{out}. The edge set E is an acyclic set of directed edges; each edge is either a pair (v_i^{out}, w_j^{in}), or a pair (c, w_j^{in}) for some $c \in \mathcal{U}$. The vertex set V is partitioned into five disjoint subsets:

V_i. a singleton containing the *input vertex* a, with $\text{ins}(a) = 0$.

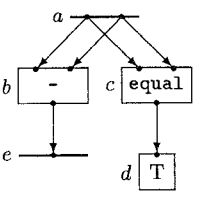
V_o. a singleton containing the *output vertex* z, with $\text{outs}(z) = 0$.

V_T. a set of zero or more vertices called *true predicates*. For each $v \in V_T$, $\text{ins}(v) = 1$ and $\text{outs}(v) = 0$.

V_F. a set of zero or more vertices called *false predicates*. For each $v \in V_F$, $\text{ins}(v) = 1$ and $\text{outs}(v) = 0$.

V_C. a set of zero or more vertices. For each $v \in V_C$, $\text{ins}(v) \geq 1$ and $\text{outs}(v) \geq 1$.

[5]From Feynman's Principle of Least Action [Feynman et al. 1964].



$$
\begin{aligned}
V_i &= \{a\} \\
V_o &= \{e\} \\
V_T &= \{d\} \\
V_F &= \{\} \\
V_C &= \{b, c\} \\
E &= \{(a_1^{out}, b_1^{in}),\ (a_1^{out}, c_1^{in}), \\
&\quad (a_2^{out}, b_2^{in}),\ (a_2^{out}, c_2^{in}), \\
&\quad (b_1^{out}, e_1^{in}),\ (c_1^{out}, d_1^{in})\}
\end{aligned}
$$

Fig. 8. Example of a simple trace graph.

The function label : $(V_C \cup V_T \cup V_F) \rightarrow L$ assigns to each vertex except V_i and V_o a label from some label set L.

To simplify notation from this point on, we will write trace graphs as $G = (V, E)$, treating the vertex arities and labels as vertex properties implicit in V.

Figure 8 shows a simple example of a trace graph. Observe that the input and output vertices are drawn as horizontal lines, the predicates as squares, and the other vertices as rectangles; also, the vertices are marked with dots to show their input and output arities.

Graphs that have some structure in common will play a key role in later discussions, so we will make use of the following definition:

Definition 3.1.1.2. Vertex v in trace graph G *matches* vertex v' in trace graph G' if and only if the subgraph of G consisting of v and all its ancestors is isomorphic to the subgraph of G' consisting of v' and all its ancestors.

In this and all subsequent appeals to trace graph isomorphism, it is assumed that isomorphism considers vertex arities, partitions, and labels as well as edge structure, so that isomorphic trace graphs are identical up to vertex renaming.

3.1.2 *Vertex Labels and Executions.* Each vertex in $V_C \cup V_T \cup V_F$ has a vertex label. Ordinarily this label is a relation on the universe of values $\mathcal{U}$. All vertices $v \in V_T$ have label$(v) = \{(\text{true})\}$ (a unary relation, since the input arity is 1, and the output arity is 0) and all vertices $v \in V_F$ have label$(v) = \{(\text{false})\}$. Most vertices in V_C are labeled with relations too: these are the *terminal* vertices. (When we introduce trace grammars we will find another use for the labels of *nonterminal* vertices.) In a *terminal trace graph*, all vertices are terminal, so all labels are relations of the appropriate arities.

In fact, these relations are usually functions: for any vertex v, for any tuple x of input values, there is usually a unique tuple y such that $x \cdot y \in$ label(v). For now we will assume that the label of a vertex in V_C is a function, and we will draw trace graphs by naming the function in question.

A trace graph can be thought of as a partial program that computes a function for a limited set of inputs.

Definition 3.1.2.1. Given a fully terminal trace graph $G = (V, E)$, an *execution of* G is a function f that assigns a value from $\mathcal{U}$ to each vertex input and output, such that:

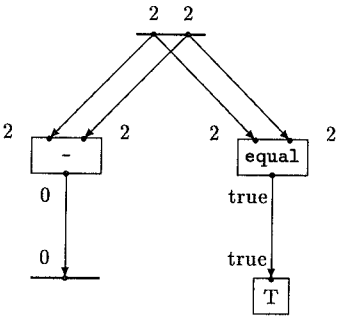


Fig. 9. Execution of a trace graph on (2,2).

—For each edge (v_i^{out}, w_j^{in}), $f(w_j^{in}) = f(v_i^{out})$; for each edge (c, w_j^{in}), $f(w_j^{in}) = c$.
—All $v \in V_C \cup V_T \cup V_F$ with input arity n and output arity m satisfy

$$(f(v_1^{in}), ..., f(v_n^{in}), f(v_1^{out}), ..., f(v_m^{out})) \in \text{label}(v).$$

If x is the tuple of values assigned by an execution f to the graph inputs, we will say that G has execution f on x. The *domain* of a graph, $\text{dom}(G)$, is the set of input tuples for which G has an execution.

Figure 9 shows an execution of a trace graph. (This graph has an execution on an input (n, d) if and only if $n = d$.) The important thing about executions is that they fix not only the relation of inputs to outputs, but also the relation of inputs to all intermediate values. This relation is, in fact, a partial function: for any input there is at most one execution.

3.1.3 *Thinning Trace Graphs.* The PLC sanctions the removal of unnecessary computations from a program, which corresponds to the removal of vertices from a trace graph. How can you remove some vertices and still have a legal trace graph? You would have to remove not only the vertices in question, but all the edges to and from those vertices. You would probably also have to add some edges. Since every vertex input in a trace graph must be the target of exactly one edge, you would have to add edges to *bypass* the gap, so that every vertex that used to get its input from one of the excised vertices would be the target of an edge from elsewhere in the graph.

Definition 3.1.3.1. Let $G = (V, E)$ be a trace graph, W a vertex set with $W \subseteq (V_C \cup V_T \cup V_F)$. Let E_W be that subset of E with source or destination vertices in W. A *bypass* for W is an edge set F such that $((V \setminus W), (E \setminus E_W \cup F))$ is a legal trace graph.

A bypass is simply a set of new edges; it has an edge for every edge that used to leave W, supplying the same target vertex but from a source that is not a vertex in W. It does so in a way that makes $((V \setminus W), (E \setminus E_W \cup F))$ satisfy the definition of a trace graph; in particular, the new edge set must be acyclic.

The idea of removing and bypassing some vertices is critical. We call this transformation a *thinning*. It induces a "thinner-than" relation $\sqsubseteq$ preserved by isomorphism: one graph is thinner than another if it is isomorphic after removing and bypassing a subset of vertices.

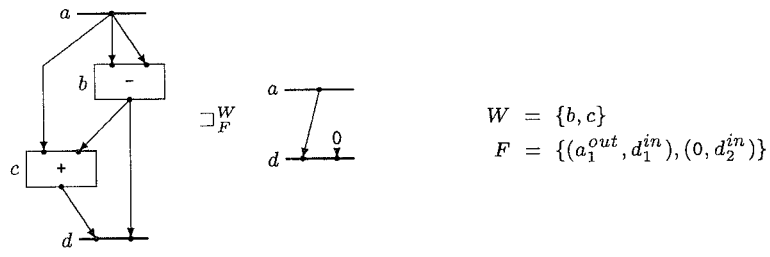


Fig. 10. A thinning.

Definition 3.1.3.2. Let $G = (V, E)$ and $G' = (V', E')$ be trace graphs. $G' \sqsubseteq G$ if and only if there exists some vertex set $W \subseteq (V_C \cup V_T \cup V_F)$ and some bypass set F such that $((V \setminus W), (E \setminus E_W \cup F))$ is isomorphic to G'. $G' \sqsubset G$ if and only if $G' \sqsubseteq G$ and G' is not isomorphic to G.

To identify the W and F involved in the thinning, we will sometimes write $G' \sqsubseteq_F^W G$. Figure 10 shows an example of a thinning.

Usually, when you thin a graph, you end up with a graph that computes some completely different (partial) function. That is not going to be useful in an optimizer! We are really interested in a special class of thinnings: those that preserve functionality in a particular way. The two graphs of Figure 10 already stand in this (much more demanding) relation. Executions of the two graphs on the same value always agree.

Definition 3.1.3.3. Suppose $G = (V, E)$ and $G' = (V', E')$ are terminal trace graphs. $G' \underline{\sqsubseteq} G$ if and only if:

(1) $G' \sqsubseteq G$. (This defines an injective function h from vertex inputs and outputs of V' into corresponding vertex inputs and outputs of V.)
(2) For every execution f of G there is an execution f' of G' such that $f' = f \circ h$.
(3) For every execution f' of G' there is an execution f of G such that $f' = f \circ h$.

If $G' \sqsubset G$ in condition 1, $G' \underline{\sqsubset} G$.

This is *conservative thinning*: thinning that removes vertices from a graph in a way that preserves the values assigned to the remaining vertices by all executions.[6] This is much stronger than saying that the function is extensionally equivalent after thinning: not only the function's outputs, but indeed all surviving intermediate values, are preserved. Observe, by Definition 3.1.3.3, that predicates are only unnecessary if they play no role in limiting the set of inputs for which the graph has an execution. If removing a predicate would allow additional executions, enlarging the domain of the graph, the thinning is not conservative.

3.1.4 *Discussion.* The trace graph can be thought of as a partial evaluation of a program: a partial evaluation with respect to a fixed control path as in the work of

[6] Johnson [1986] uses the term "thinning" in a different context, to refer to any restriction of an equivalence relation that retains at least one member of each equivalence class. This is related to our "conservative thinning" only accidentally; see the proof of Theorem 3.1.4.3, below.

Perlin [1989]. A trace graph is a partial program which is equivalent to some full program on a subset of inputs (those for which it has an execution). By thinning it conservatively we can optimize it for operation on such inputs.

Discovering violations of the PLC appears to be simply a matter of finding conservative thinnings. Unfortunately, the word "simply" is something of an overstatement: the question of whether a thinning is conservative is undecidable even for a very restricted class of trace graphs.

THEOREM 3.1.4.1. *Consider the class of trace graphs containing only two-input integer additions, multiplications and exponentiations, positive integer constants, and integer equality tests. The question of whether a thinning is conservative is undecidable for this class.*

PROOF. This follows from the existence of an exponential diophantine equation with one parameter N which is unsatisfiable if and only if the program with Gödel number N does not halt on any input [Jones and Matijasevic 1984]. □

This is our first glimpse of the inference problem Thinner will have to deal with: it must perform some kind of semantic analysis on trace graphs to identify conservative thinnings. It must also carry out any conservative thinnings it identifies. Suppose we have all the semantic analysis we need: a partition of the vertex inputs and outputs into equivalence classes (so that two objects are in the same equivalence class if and only if they are assigned equal values in all executions) and a constant for each constant equivalence class. Can we then thin the graph optimally and efficiently? The answer is no: the problem of identifying an optimal thinning is intractable.

Definition 3.1.4.2. The *minimum-thinning problem* is this: given a trace graph $G = (V, E)$ and an integer $k \leq |V|$, and given a partition of the vertex inputs and outputs into equivalence classes, decide whether there is a graph $G' = (V', E')$ such that $G' \sqsubseteq G$; the thinning is conservative with respect to the given partition; and $|V'| \leq k$.

THEOREM 3.1.4.3. *The minimum-thinning problem is NP-complete.*

PROOF. The minimum-thinning problem is in NP, since we can verify in polynomial time whether a particular G', W, and F satisfy $G' \sqsubseteq_F^W G$, and whether that thinning is conservative with respect to the given partition. The remainder of the proof is by transformation from minimum cover [Garey and Johnson 1979].

An instance of the minimum-cover problem is given by a collection C of subsets of a finite set S along with a positive integer $k \leq |C|$. We will construct a corresponding instance of the minimum-thinning problem as follows. Let there be one equivalence class E_s for each element $s \in S$. For each $c \in C$, let there be a 1-input, $|c|$-output vertex $v \in V_C$; and for each $s \in c$ assign one vertex output v_i^{out} to the equivalence class E_s. Let the graph have a single input, with an edge to the input of each $v \in V_C$; and let it have $\sum_{c \in C} |c|$ outputs, one for each vertex output in V_C.

Now for a given k, there is a $G' = (V', E')$ such that $G' \sqsubseteq G$; the thinning is conservative with respect to the partition; and $|V'| \leq k$—if and only if there is a $C' \subseteq C$ such that every element of S belongs to at least one member of C' and $|C'| \leq k$. This follows immediately since vertices can be removed in a conservative thinning if and only if a representative of every equivalence class remains. □

This proof involves the construction of trace graphs with unbounded degree, which may seem unreasonable. But with a bit more work one can prove the same result for graphs with vertices restricted to indegree 1 and outdegree ≤ 3. (Note that the minimum-cover problem remains NP-complete even if all $c \in C$ have $|c| \leq 3$.)

So even if the semantic analysis problem is licked, the problem of choosing an optimal thinning is still intractable. These are two problems that Thinner faces: identifying violations of the PLC and repairing them. But the uncomputability of the first and the intractability of the second turn out to have little further significance. In practice, the inference problem is not the problem of finding all violations, but the problem of finding as many violations as possible, as quickly as possible. The reformulation problem involves deep difficulties in transforming recursive structures (the grammar-thinning problem), and we will gladly settle for a greedy heuristic when we finally get around to thinning individual trace graphs.

But this is a digression from our immediate goal, which is to develop a formalism for complete programs.

3.2 Sets of Trace Graphs

As a single trace graph represents a path of execution in a functional program, so a set of trace graphs represents a collection of paths. If constructed correctly, a set of trace graphs may represent exactly the collection of all possible paths for a single program. The examination of these *executable sets* is the next step in the development of our formalism for programs: at this level we will take conditional execution into account, but still defer issues relating to recursive structures in the source language.

3.2.1 *Sets of Trace Graphs.* To represent a program, a trace graph set must meet several conditions. Of course, the extents of V_i and V_o must match for all graphs in the set; from now on we will make this assumption about all trace graph sets, and we will use $\mathrm{dom}(S)$ to denote $\bigcup_{G \in S} \mathrm{dom}(G)$. The set must also be deterministic, in the sense that it must not have more than one member with an execution for a given input. But these two properties by themselves do not guarantee that a set of trace graphs corresponds to a program. For example, consider the set shown in Figure 11. It has exactly one graph with an execution for any input, but the difference between the two graphs in the set represents a choice that must be made by any corresponding program, and there is no predicate that can be evaluated in both traces in time to guide the choice. Any program with these two traces would have to be prescient—it would have to decide immediately and spontaneously whether to compare or to subtract.

Imagine that you are given a deterministic set of trace graphs and an appropriate input vector. Your task is to perform exactly those computations called for by the trace graph with an execution for that input, if any. *You are not told which graph in the set actually applies to that input*, so the trick is to narrow the set of possibilities by evaluating predicates and discarding those graphs that do not have an execution; for the task to be possible, there must be a way to do this without deviating from the computation called for by every trace graph not yet eliminated. When this is possible for every input, the set of trace graphs is called *executable*.

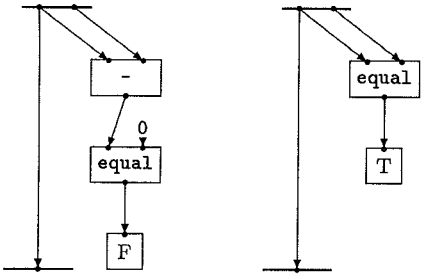


Fig. 11. A nonexecutable set of trace graphs.

To make this concept more precise, we first define a *predicate partition* of a set of trace graphs, which splits a set in two using a predicate common to all graphs in the set. Then we define an executable set of trace graphs inductively: a set is executable if it is a singleton or empty or if it can be predicate partitioned into executable sets.

Definition 3.2.1.1. A *predicate partition* of a set S of trace graphs is a function p identifying a predicate of each element of S, such that for any G and G' in S, $p(G)$ matches $p(G')$ (except that one may be a true predicate and the other a false predicate).

So a predicate partition selects a predicate (which is just a vertex in V_T or V_F) from each graph in S. These predicates match each other, which is to say the subgraph of ancestors of a predicate in one graph is isomorphic to the subgraph of ancestors of the predicate in another. The predicates thus represent true-or-false decisions which are computed identically in every graph in S, and the predicate partition divides S into two sets: those for which the predicate is true and those for which it is false.

Definition 3.2.1.2. A *decision tree* for a set S of trace graphs is a binary tree with a subset of S at each node and a predicate partition of that subset at each internal node. If S is empty or is a singleton, the decision tree for S is a leaf giving S. Otherwise, a decision tree for S is a node giving both S and a predicate partition for S, whose left child is a decision tree for that subset of S for which the predicate is true, and whose right child is a decision tree for that subset of S for which the predicate is false. No predicate can occur more than once in any partition in the tree.

Definition 3.2.1.3. An *executable* set of trace graphs is one for which there is a decision tree.

A decision tree is a tree of if-then-else refinements, zeroing in on a trace graph with an execution for a given input. Clearly any program defines a decision tree and must generate an executable set of trace graphs. The next theorem shows that an executable set can be thought of as a kind of program itself, since it contains exactly one trace graph with an execution for any input.

THEOREM 3.2.1.4. *If S is an executable set of trace graphs, and if all vertex labels are functions, then for any input vector x there is at most one $G \in S$ with an execution on x.*

PROOF. Suppose S is executable, and suppose by way of contradiction that there are two different graphs G and G' in S with executions f and f' on the same input vector x. S has a decision tree with G and G' in different leaves: let N be that common ancestor of G and G' furthest from the root of that decision tree. At N there is a subset of S containing G and G' and a predicate partition p that selects a true predicate for one and a false predicate for the other. Without loss of generality, assume $p(G) = v \in V_T$ and $p(G') = v' \in V'_F$. Then $f(v_1^{in}) =$ true and $f'(v_1'^{in}) =$ false. But v and v' are matching vertices, so $f'(v_1'^{in}) = f(v_1^{in})$. This is a contradiction. □

Theorem 3.2.1.4 shows that any executable set is deterministic. As Figure 11 shows, the reverse is not true.

3.2.2 *The Thinning Relations for Trace Graph Sets.* The idea of thinning developed for individual trace graphs extends naturally to sets of trace graphs. One set of trace graphs is thinner than another when every member of the thinner set is thinner than some member of the thicker set.[7]

Definition 3.2.2.1. Suppose S and S' are sets of trace graphs. $S' \sqsubseteq S$ if and only if for every $G' \in S'$ there is some $G \in S$ for which $G' \sqsubseteq G$. $S' \sqsubset S$ if and only if $S' \sqsubseteq S$ and there is no one-to-one mapping of graphs in S' to isomorphic graphs in S.

The $\sqsubseteq$ relation also generalizes to trace graph sets, but more care is required. Obviously, a conservatively thinner set should be, structurally, a thinner set; as with individual trace graphs, the additional constraint has to do with agreement between executions. A trace graph set is a function just like an individual trace graph (except that it may be nondeterministic). For one set to be conservatively thinner than another, the two sets must have the same domain, and every execution of a graph in the thinner set must agree with some execution of a graph in the thicker set.

Definition 3.2.2.2. Suppose S and S' are sets of trace graphs. $S' \mathrel{\underline{\sqsubseteq}} S$ if and only if:

(1) $S' \sqsubseteq S$.
(2) For every execution f of some $G = (V, E) \in S$ there is an execution f' of some $G' = (V', E') \in S'$ such that $G' \sqsubseteq G$ (this defines an injective function h from vertex inputs and outputs of V' into corresponding vertex inputs and outputs of V) and $f' = f \circ h$.
(3) For every execution f' of some $G' = (V', E') \in S'$ there is an execution f of some $G = (V, E) \in S$ such that $G' \sqsubseteq G$ (this defines an injective function h from vertex inputs and outputs of V' into corresponding vertex inputs and outputs of V) and $f' = f \circ h$.

If $S' \sqsubset S$ in condition 1, $S' \mathrel{\underline{\sqsubseteq}} S$.

Note that this definition gives $\{G'\} \mathrel{\underline{\sqsubseteq}} \{G\}$ if and only if $G' \mathrel{\underline{\sqsubseteq}} G$, as expected. The definition does not require that every graph in the thicker set be thicker than some graph in the thinner set. If a graph in the thicker set has no executions, it need

[7] Compare this with the lower or Hoare powerdomain [Gunter and Scott 1990].

not be represented in the thinner set. The fact that the $\sqsubseteq$ relation allows the removal of vacuous graphs from the set as well as conservative thinnings of graphs within the set is important because it admits a significant class of "dead-code" transformations.

If $S' \sqsubseteq S$ and if S is executable, it does not necessarily follow that S' is executable: each graph in S may be thinned in a different way, so that there are no matching predicates for a decision tree.

3.2.3 *Discussion.* An executable set of trace graphs is a program expressed without recursion: every possible path is explicit. This is part of what makes this a good domain for the formal expression of the PLC.

Executable sets express a bare minimum of control: there is at least one order of execution that selects the right trace for any given input without deviating from that trace, but the executable set does not say what that order is. This may seem unnecessarily abstract, but it too contributes to clean formalization of the PLC. Consider an alternative formalism, the dynamic dataflow graph in the style of Arvind and Nikhil [1987], in which the two arms of a conditional have inputs directed to them by "switch" nodes and outputs collected from them by "mux" nodes.

The code fragment of Figure 12 exposes an opportunity for common-subexpression elimination: the evaluation of g is redundant if the left branch is taken. The code motion required by this transformation in the textual and dataflow cases is an artifact of the representations; it is necessary because those representations make a commitment to the exact point at which conditional parts of the computation diverge. The executable set representation makes it clear that this optimization merely removes some intermediate values without affecting the result: in short, it is a thinning.

3.3 Trace Grammars

Most interesting programs have infinitely many possible paths of execution, so the explicit trace graph set is going to be inadequate as a representation. What we need is a finite representation that implicitly identifies a potentially infinite trace graph set. We therefore turn to trace grammars, which are finite objects that generate sets of trace graphs.

3.3.1 *Trace Grammars.* Trace grammars will provide a mechanism for deriving one trace graph from another, and so for generating a language of trace graphs from an original "start graph." In a trace grammar one trace graph derives another by vertex replacement. The mother vertex and the daughter graph with which it is to be replaced must have matching input and output arities. Remove the mother vertex from the host graph, and remove the input and output vertices from the daughter graph; then stitch the daughter graph into the host graph with a "seam" of edges connecting the matching inputs and outputs. This seam of edges includes one edge to each vertex input left sourceless in the daughter graph (those that used to have an edge from the input vertex of the daughter graph), and one to each vertex input left sourceless in the host graph (those that used to have an edge from the mother vertex). Formally:

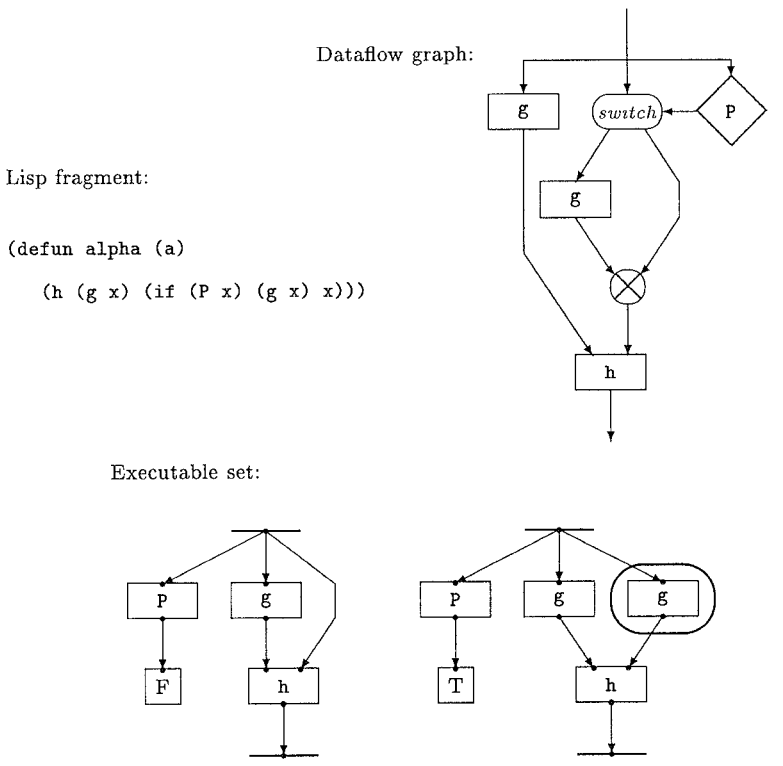


Fig. 12. A comparison of three representations.

Definition 3.3.1.1. Let $G = (V, E)$ and $H = (W, F)$ be trace graphs with disjoint vertex sets, and let v be a vertex in V_C matching H in input and output arity. Define E_{old} to be E restricted to $V \setminus \{v\}$, and define E_{new} to be F restricted to $W_C \cup W_T \cup W_F$. Define E_{seam} as follows:

—For each edge in F from the jth vertex output of W_i to some w_k^{in}, E_{seam} has an edge (u_i^{out}, w_k^{in}), where u_i^{out} was the source of the edge in E to the jth vertex input of v.

—For each edge from v, $(v_j^{out}, u_k^{in}) \in E$, E_{seam} has an edge (x_i^{out}, u_k^{in}), where x_i^{out} was the source of the edge in F to the jth vertex input of W_o.

The *replacement of v in G by H* yields a trace graph $G' = (V', E')$, where $V' = V \setminus \{v\} \cup W_C \cup W_T \cup W_F$ and $E' = E_{\text{old}} \cup E_{\text{seam}} \cup E_{\text{new}}$.

Definition 3.3.1.2. A *trace grammar* is a 4-tuple (N, T, P, S). N is a finite set of *nonterminal labels*, and T is a finite set of *terminal labels*. (N and T are disjoint.) P is a finite set of *productions* of the form $(\alpha \rightarrow D)$, where $\alpha \in N$, and D is a trace graph with the same input and output arities as α. S is a graph called the *start graph*. All of the V_C, V_T, and V_F vertices in P and S have labels in $N \cup T$.

Suppose that $Q = (N, T, P, S)$ is a trace grammar and $G = (V, E)$ and $G' = (V', E')$ are trace graphs in which all of the vertices have functions in $N \cup T$. Then $G \Rightarrow_Q G'$ if and only if there is a vertex $v \in V_C$ with a nonterminal label α, and a production $(\alpha \rightarrow D)$ in P, such that the replacement of v in G by a graph isomorphic to D that shares no vertices with G yields G'. This is the mechanism by which one graph is derived from another directly. It can be extended in the usual way to the reflexive and transitive closure: $G \Rightarrow_Q^* G'$ if and only if there is a sequence of zero or more single-step derivations from G to G'. The $\Rightarrow_Q^*$ relation can be used in the usual way to define the set of trace graphs generated by a trace grammar.

Trace grammars are a specialization of the directed, node-label controlled (DNLC) graph grammars studied by Rozenberg [1987], structurally (though not semantically) similar to Feder's plex grammars [Bunke and Haller 1990; Feder 1971]. Not all trace grammars generate executable sets; but there is a restriction on the set of productions that guarantees that the generated set will be executable.

Definition 3.3.1.3. A *normal* trace grammar $Q = (N, T, P, S)$ is one in which all nonterminals α in N have the property that $\{D \mid (\alpha \rightarrow D) \in P\}$ is an executable set.

THEOREM 3.3.1.4. *If Q is a normal trace grammar, $L(Q)$ is executable.*

PROOF. Let R be the set generated by a normal trace grammar $Q = (N, T, P, S)$. Define a function f that maps each graph $G \in R$ to a breadth-first derivation of G.[8] We will define a decision tree $T_f(R)$ recursively using the breadth-first derivations chosen by f. Let n be the greatest integer with the property that all $f(G)$ for $G \in R$ agree in the first n steps. Since f chooses a fixed-order derivation, all the $(n+1)$th steps are expansions of the same nonterminal, and since f chooses a breadth-first derivation, that nonterminal has terminal ancestors only. So $T_f(R)$ can start with a decision tree for the possible expansions of that nonterminal, which must exist since Q is normal. This partitions R into subsets that agree on the first $n+1$ steps; for any such $R' \subset R$ with more than one member, construct a decision tree $T_f(R')$ recursively. Since all $f(G)$ for $G \in R'$ agree on the first $n+1$ steps, $T_f(R')$ will only use predicates generated by nonterminals that were not yet expanded when $T_F(R)$ was constructed. It follows that no predicate is used more than once, so the resulting structure is a decision tree for R. □

3.3.2 *Correspondence to CFGs.* Can any executable set be generated by some trace grammar? To answer this question, we developed the connection between trace grammars and traditional context-free grammars on string alphabets [Webber 1993]. The answer is no: a language of graphs need not be "context free" to be executable. A reduction from CFG questions to trace-grammar questions can also be used to show that many questions about trace grammars are undecidable. Among these are the following:

—Are the sets generated by two trace grammars disjoint?

—Do two trace grammars generate isomorphic sets?

[8] A trace graph does not necessarily have a unique breadth-first derivation. If this troubles you, derivations can be ordered and the least breadth-first derivation selected for each graph.

—Is the set generated by one trace grammar a subset of the set generated by another?

—Is there a trace grammar for the intersection of the sets generated by two trace grammars?

The reader should observe that the undecidability of these questions does not follow immediately from Rice's Theorem. There is a difference between these questions, which have to do with the recursive structure of a program, and questions involving the actual behavior of programs on inputs. For example, it is decidable whether the set generated by a trace grammar is empty, and if it is empty the program does not halt. But it does not follow that we can decide whether the program halts, because it may be the case that the generated set is not empty, but none of the trace graphs in it has an execution on any input.

Consider how a trace grammar expresses nontermination. Suppose a program allows infinitely many different paths of execution. It must be possible for such a program to diverge (by König's lemma). The corresponding trace grammar generates an infinite set of trace graphs, each of which is a finite structure representing a terminating execution history. *No infinite graph is generated by the grammar*; there is no trace graph with an execution for those inputs on which the program diverges. Suppose a program allows only finitely many paths of execution yet still diverges on some input. This is possible if one of those "paths of execution" is a branchless infinite loop. The corresponding trace grammar supports an infinite chain of derivations, but that chain never produces a fully terminal trace graph; once again, no infinite graph is generated, and there is no trace graph for those inputs on which the program diverges. A trace grammar for a program that diverges on an input generates an executable set containing no graph with an execution for that input—an executable set with a decision tree that is not a full binary tree.

3.3.3 *Thinning and Trace Grammars.* The $\sqsubseteq$ and $\sqsubseteq_c$ relations extend to graph grammars in the obvious way: one grammar is thinner than another when the language it generates is thinner.

Definition 3.3.3.1. The relations $\sqsubseteq$, $\sqsubset$, $\sqsubseteq_c$, and $\sqsubset_c$ hold for trace grammars if and only if they hold for the sets generated by those grammars.

This definition is not very satisfying, since it refers again to those unwieldy infinite sets. It is not an effective definition, since it cannot be used to derive thinner grammars or even to test whether one grammar is thinner than another. It would be so much more practical to have an effective characterization of $\sqsubseteq$ and $\sqsubseteq_c$ for grammars, in terms of the grammars themselves. Is such a thing possible?

An effective characterization of $\sqsubseteq_c$ for grammars is clearly not possible. Theorem 3.1.4.1 shows that the $\sqsubseteq_c$ relation is undecidable even for individual trace graphs, and with trace grammars there are, in addition, all the usual undecidabilities of fully expressive formalisms for computation. For example, a grammar generating the empty set is conservatively thinner than a grammar generating a set S if and only if $\mathrm{dom}(S) = \emptyset$, so an effective characterization of $\sqsubseteq_c$ would decide the halting problem.

What about $\sqsubseteq$? It is a relatively simple relation on graph structures and makes no reference to executions. Is the $\sqsubseteq$ relation on trace grammars decidable? We do

not know. However, there is an interesting corresponding question about CFGs. Define $L(C)$ to be the language of a CFG C, and define $\text{Omit}(L)$ to be the language of strings that can be formed by dropping 0 or more characters in any positions from any string in language L. Is it decidable whether $L(C) \subseteq \text{Omit}(L(C'))$? Using the connection between CFGs and trace grammars described above, this question can be reduced to the question of whether the trace grammar corresponding to C is thinner than the trace grammar corresponding to C', so if it were undecidable, the $\sqsubseteq$ relation for trace grammars would also be undecidable. Surprisingly, it is decidable: $\text{Omit}(L(C'))$ is a regular language.[9] So perhaps the $\sqsubseteq$ relation is decidable too.

We can now express the Principle of Least Computation formally: for an executable trace grammar Q there should be no executable $Q' \underline{\sqsubset} Q$. A program is a trace grammar that generates a possibly infinite executable set of trace graphs. The program violates the PLC if and only if there is another program whose trace grammar generates a conservatively thinner executable set.

3.4 Applying the PLC

Now we have a formal statement of the PLC. How can it be embodied in an automatic optimizer? We cannot just search for conservatively thinner grammars! One observation is that if a grammar violates the PLC for trace grammars, it generates some graph that violates the PLC for trace graphs. (That is, if a program violates the PLC then there is some path through the program that makes an unnecessary computation.)

Thinner makes use of this. It first searches a space of graphs (not necessarily terminal) generated in the grammar, looking for violations of the PLC. If it finds such a graph, Thinner uses it as a thinning example: it tries to reformulate the program so that it never makes the mistake illustrated in the example. This step, using the method of kernels, involves reorganizing the grammar so that thinnable graphs occur explicitly in the right-hand sides of productions. The actual thinning step is then performed on the set of right-hand sides for a given nonterminal. This is done in such a way that the set remains executable (and the grammar remains normal).

There is a theoretical impediment to all this, though: graphs in the right-hand sides of productions in a grammar may contain nonterminals, but we only defined the $\underline{\sqsubset}$ relation for fully terminal graphs. It could be defined with reference to expansions of the nonterminals, but this leads back to infinite sets. To localize the problem we must treat nonterminals more like terminals, without investigating how they expand.

The solution is to test $\underline{\sqsubset}$ relative to a mapping from nonterminals to partial functions. If H and G are trace graphs containing nonterminals, and if $\mathcal{F}$ is a function mapping each nonterminal to a function, and if that mapping gives $H \underline{\sqsubset} G$, we will say that $H \underline{\sqsubset} G$ *relative to* $\mathcal{F}$.

This raises an important point: there are two approaches to thinning the right-hand sides of productions. Strictly speaking, one should assume nothing about

[9] A neat proof that $\text{Omit}(L(C'))$ is regular was suggested by Juris Hartmanis. The key step is to express $L(C')$ as $h(L_D \cap R)$ for a homomorphism h, a Dyck language L_D, and a regular language R, which is possible for any CFG by a result of Chomsky [1962].

$\mathcal{F}$: any nonterminal can be any partial function. Conservatively thinning a graph containing partial functions is tricky because partial functions limit the class of inputs for which the graph has executions: if you remove one in a thinning, you may be adding executions, which would be nonconservative. Since the domain of the arbitrary function associated with a nonterminal is unknown, a conservative thinning must retain at least one computation of each nonterminal function on each distinct input value. Treating nonterminal sets of trace graphs in this way is a safe approximation to full conservative thinning: it misses some cases but never makes a false step.

The other approach is to treat each nonterminal as an arbitrary, deterministic, *total* function. This *weak conservative thinning* is discussed more formally elsewhere [Webber 1993]. It is the approach Thinner takes—it just does not make sense to waste any computational effort on ensuring that the set of inputs on which the program diverges does not shrink.

4. THINNER

We now turn to practical matters: the implementation of Thinner, an optimizer based on grammar thinning. Thinner uses the trace grammar formalism developed in Section 3 as its intermediate representation. It compiles its input program into a trace grammar, reasons about the grammar to identify a weak conservative thinning example, applies the method of kernels from Section 2 to reformulate the grammar with respect to that thinning example, and finally decompiles the grammar back into the source language.

The part of this process that has not been discussed in this article is the inference step: how can weak conservative thinnings of a trace graph be detected? As mentioned above, these are just conservative thinnings, plus thinnings that eliminate unnecessary nonterminals even when doing so enlarges the graph's domain. And finding conservative thinnings is, by Theorem 3.1.4.1, an undecidable inference problem. The construction of Thinner demanded a practical approach to this problem: a technique for finding as many violations of the principle as possible, as quickly as possible. For this purpose we developed a new inference method called *relational constraint*, which is discussed elsewhere [Webber 1992].

This section presents a summary of the design and implementation decisions that went into the construction of Thinner and discusses some of the problems of translating the theoretical tools developed in the previous sections into a working system. It includes examples of optimizations performed by Thinner.

4.1 Compilation and Decompilation

The source language optimized by Thinner is called TL (naturally, Thinner's Language). TL is a purely functional, strongly typed first-order language, with several scalar types and corresponding flat list types. A formal semantics for TL [Webber 1993] elucidates the connection between this language and trace grammars: each TL expression denotes a trace graph and each function definition, a trace grammar. This makes compilation straightforward.

The process of finding and correcting violations of the PLC is an iterative one: the more time Thinner has, the more optimizations it may discover. At any time the user of Thinner can ask to see a decompiled version of the current grammar.

Decompilation is more difficult than compilation, but the grammar transformation step guarantees that the current grammar is always normal (Definition 3.3.1.3), which means that decompilation is always possible.

Because the grammar is normal, each nonterminal can be decompiled independently. Decompilation of a nonterminal is guided by a decision tree (Definition 3.2.1.2) for the set of right-hand sides for that nonterminal. There may be more than one decision tree, and Thinner collects them all. Each decision tree would yield a slightly different (though grammatically and functionally equivalent) decompilation; Thinner selects the first one it finds to guide the decompilation of the nonterminal. (There is another use of decision trees in Thinner, for which it may need them all: the computation of thinning obligations, discussed in Section 4.3.)

4.2 Searching for Thinning Examples

Using depth-first iterative deepening, Thinner explores the space of graphs that can be derived from nonterminals in the grammar. This is not systematic: it will sometimes develop the same graph along more than one path. It is not particularly clever either.

Only one heuristic is employed to guide the search toward significant thinning examples: Thinner unfolds graphs in order, treating those with the most nonterminals first. (That is, it examines paths starting from graphs with the most nonterminals, and it unfolds using productions with the most nonterminals.) This helps Thinner optimize recursive cases, which can be significant improvements, ahead of base cases, which tend to be minor.

4.3 The Method for Kernels for Trace Grammars

We sketched in Section 2.5 how one can apply the method of kernels to trace grammars by generalizing each step from strings to graphs. But this generalization is far easier to imagine than to implement.

4.3.1 *Graph and Subgraph Matching.* The first hurdle is graph matching. The method of kernels uses string matching in several places. For example, the thinning step repeatedly finds and thins an instance of a string δ in a target string. The graph counterpart, naturally, is repeatedly finding and thinning an instance of a graph δ in a target graph. But this is subgraph matching, which is NP-complete: potentially, a far more expensive operation than substring matching. Another example is the test Kernelize makes to determine whether the new nonterminal X_β has already been created for a particular string β. In the graph universe, we need some kind of dictionary we can use to look up the *graph* β.

Although subgraph matching is NP-complete, Thinner's application of it is surprisingly inexpensive in practice. Trace graphs have several features that contribute to this: they are dags; they have low degree (most vertices have outdegree 1); and the vertices are labeled with functions that must match. We do not know whether the subgraph matching problem for trace grammars *generated from TL programs* is actually NP-hard.

4.3.2 *Thinning Right-Hand Sides.* A thinning example for a graph can often be thinned in more than one way. Frequently an example will turn up in which two vertex outputs are equivalent. Which one should be removed and which one

retained? Luckily, the decision can easily be postponed. The method of kernels works to make an example graph explicit in the grammar: how that graph is altered once it surfaces as the subgraph of some right-hand side of a production is unconstrained. (For this reason the method of kernels might be useful with transformations other than thinning.)

A thinning example for Thinner is a graph, together with a nonempty list of potential thinnings of that graph. When the thinning example matches a subgraph of some graph in the grammar, the individual thinnings for that example are weighed in context. The weight assigned to a thinning depends on three criteria, given in order with the most important first:

(1) How many reuses of previously created nonterminals X_β does the thinning yield?
(2) How many nonterminal vertices does it eliminate?
(3) How many terminal vertices does it eliminate?

These three criteria are combined into a numeric weight in such a way that less significant criteria can affect the comparison of two thinnings only if they are equal in terms of more significant criteria. The weight assigned to a thinning is heuristic. In fact, there is no exact way to compare two thinnings, since a thinning may affect only unreachable parts of the computation and since there is no sure way to detect this.

Eliminating primitive computations is the ultimate goal, so the least significant component of the weight is obviously worth checking. The second component, the number of nonterminals eliminated, should clearly have even greater significance: it is a good bet (though not certain) that each nonterminal eliminated saves many primitive computations. But what about the most significant part of the weight—how can the reuse of previously created nonterminals be important?

In the field of transformational optimization it is well known that "folding against a eureka definition" can be a major source of improvement; but for those not familiar with this an example may help. Consider the Fibonacci example from Section 1.1. In that example, we pretended that there was only one way to thin the example in Figure 1. In fact, there are four, as shown in Figure 13. Thinner postpones the decision about which one to use and remembers all four possibilities while applying the method of kernels.

The thinning example matches a subgraph of the productions for X_β constructed in Figure 4. At this point Thinner assigns a weight to each of the four possibilities. Without considering the question of X_β reuse, Thinner might make the wrong choice between thinnings C and D: in this context they both eliminate one terminal and one nonterminal. But thinning D is in fact vastly superior: thinning C will ultimately yield another exponential-time expression of the Fibonacci function (though with the exponent reduced), while thinning D, as we have seen, leads to the reuse of a previously created nonterminal, and so to a linear-time expression.

In short, the reuse of a previously created nonterminal X_β is desirable because it means the reuse of an *already thinned* part of the grammar.

One important difference between the graph case of the grammar-thinning problem and the string case is that in the graph case we have to make sure we end up with an executable grammar (in the sense of Definition 3.2.1.3). If we always thin

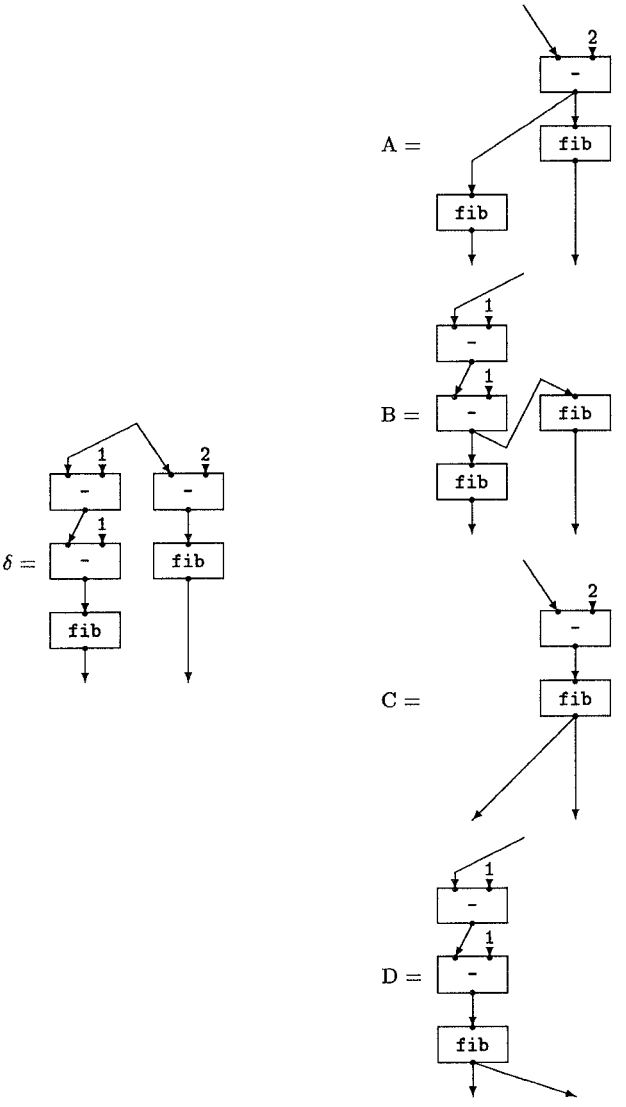


Fig. 13. Four thinning possibilities.

the right-hand sides of productions whenever we can, without regard to executability, we may end up with a trace grammar that is unexecutable—one that cannot be decompiled back into a program.

We have already observed that Thinner maintains a *normal* grammar at all times. This means that it will not thin the right-hand side of a production without first guaranteeing that there will still be a decision tree for the set of right-hand sides of that nonterminal. Usually this can be verified immediately: if a thinning does not alter any predecessor of a predicate in the graph then it has no effect on the decision trees for the set, and it can be performed with impunity. Sometimes the guarantee takes the form of "thinning obligations": by inspection of decision trees,

Thinner determines that the thinning in question can be performed only if one or more additional thinnings are performed at the same time. Occasionally, a thinning must be turned down flat: this happens when Thinner can see no way to carry out the thinning while maintaining a normal grammar.

4.3.3 *Replacing a Group of Vertices.* Another interesting complication in implementing the method of kernels for trace grammars arises at the step in which a group β is replaced by a new nonterminal X_β. In the string case we are dealing with adjacent symbols in a string, and the operation of replacing them with a single symbol is well defined. But consider the graph case. We have identified kernels in a graph, and we want two vertices to be in the same group if and only if they occur in the same instance of a kernel. But the groups constructed in this way are not necessarily replaceable by a single nonterminal vertex, at least not without introducing cycles. How Thinner deals with this problem is discussed elsewhere [Webber 1993].

4.3.4 *Constructing a Kernel Set.* The method of kernels requires a kernel set: a set of graphs that are minimal precursors of the thinning example in the grammar. Where does this kernel set come from?

It is certainly possible to compute the set of all kernels for a given grammar and example. Starting with the example, find all ways it can overlap with the right-hand side of any production in the grammar. For each such overlapping, construct the "previous" graph (the graph with the overlapped region replaced by the corresponding nonterminal), and add that to the set of kernels. Repeat the process for each new kernel until no new graphs are found.

We implemented this and found it to be quite expensive. The computation of an exhaustive kernel set of size greater than about 50 dominated the cost of grammar thinning. And it is not difficult to construct a family of examples for which the size of the exhaustive kernel set is exponential in the size of the example.

Luckily, we found that few of the kernels from the exhaustive kernel set were ever used. (That is, few actually occurred as a subgraph in some graph examined by the method of kernels.) In fact, in most cases, the number of kernel graphs actually used was exactly the depth of derivation at which the example was found—and the kernels used were exactly the minimal precursors of the example along the path of derivation on which it was discovered.

Thinner takes advantage of this. It does not construct the exhaustive kernel set, but constructs a restricted kernel set instead: the set of minimal precursors of the thinning example along the path of derivation on which it was discovered, along with other kernels from a search for derivations, performed as above but with an arbitrary depth limit. In those cases where this restricted set does not include all useful kernels from the exhaustive set, Thinner will miss some thinnings.

4.3.5 *Cleaning Up the Results.* The method of kernels may leave the program in an unattractive state. It creates a new function for every distinct grouped subgraph it encounters. Sometimes, when the group recurs after unfolding and thinning, these functions are recursive, but more often they are nonrecursive and have exactly one use. These fabricated functions may also be thinned in such a way that they wind up with unused inputs or outputs.

Thinner cleans up all fabricated functions after carrying out the method of kernels. It unfolds and eliminates all nonrecursive fabricated functions, and it eliminates any unused inputs and outputs from fabricated functions. It does not perform this cleanup on functions provided by the compiler, but only on those that were fabricated by the method of kernels.

These cleanup operations look like optimizations: in-line function expansion (or "open coding") is a traditional compiler optimization, and the elimination of unused function inputs and outputs seems like one too. But note that from Thinner's point of view these are neutral transformations. Our formal statement of the PLC does not attach any cost to a nonterminal other than the cost of the terminals into which it expands.

Actually, these operations do have an impact that is more that merely aesthetic. They affect the future behavior of the search for thinning examples by contracting the search space. This can change the order in which Thinner discovers and treats future thinning examples.

4.4 Thinner in Action

We begin with a transcript of a Thinner session. This is an exact transcript, except that some of the variable names generated by Thinner have been changed to improve readability.

```
> (suite "lir1")
Compiling test program LIR1 (NIL).

(DEFUN LIR1 (A B) (IF (< A 0) (LIR1 (+ A (* B (* B B))) B) A))

NIL
> (thin)
Accumulated optimization time: 0.27s.
NIL
> (decompile-all)

(DEFUN LIR1 (A B)
  (DECLARE (INTEGER A) (INTEGER B))
  (IF (< A 0) (LET ((C (* B (* B B)))) (F (+ A C) C)) A))
(DEFUN F (X Y)
  (DECLARE (INTEGER X) (INTEGER Y))
  (IF (< X 0) (F (+ X Y) Y) X))
NIL
>
```

The first command instructs Thinner to compile a file. Thinner reads in the contents of the file, echoes it to the screen, and compiles it into a trace grammar, which becomes the current grammar for subsequent operations. The second command tells Thinner to find one violation of the PLC and repair it. Thinner finds a violation, applies the method of kernels to eliminate it from the current grammar, and prints the elapsed time. The third command tells Thinner to decompile the current grammar; Thinner prints the resulting program. Note that Thinner performs type inference—the input language is strongly typed, but type declarations may usually be omitted. Thinner always includes type declarations when decompiling.

The optimization shown above is a functional analogue of a common compiler optimization, loop invariant removal. But there is no special handling for loops in Thinner. Thinner performs general elimination of unnecessary code using the method of kernels; sometimes this ends up looking like one of the traditional loop optimizations, but Thinner treats a loop the same as any other recursive function.

Thinner also performs common-subexpression elimination, dead-code elimination, constant folding, loop jamming, loop splitting, and code sinking. It finds all these optimizations using only grammar thinning by the method of kernels. It optimizes not just locally, not just interprocedurally, but interactivationally—across recursive invocations of the same function, as in the Fibonacci optimization. Tables I and II show, in condensed form, some additional examples of optimizations performed by Thinner. Most of the optimizations involve multiple thinnings; total elapsed time is shown. Type declarations have been removed from the output.

4.5 What Thinner Cannot Do

First of all, Thinner cannot find optimizations that are not thinnings! Low-level optimizations like register allocation, peephole optimization, and instruction scheduling are obviously out of the picture, and some high-level optimizations are not thinnings either.

Some traditional optimizations like code hoisting and strength reduction make use of a more complicated cost model than the formal PLC. Code hoisting reduces the size of a program without reducing the length of any path through it; this is not thinning since the PLC says nothing about the size of the grammar. Strength reduction is useful because some primitive operations are more efficient than others; this is not justified by the PLC, which treats all terminals as having unit cost.

Strength reduction fails to be a thinning for a more obvious reason: Thinner has no way to substitute one primitive for another. When we thin a program we remove redundant computations, but we never add, alter, or reorder the computations that are preserved. For this reason, not all algebraic optimizations are thinnings. For example, we might want to optimize `(+ (* a b) (* a c))` to `(* a (+ b c))`, but this is clearly not a thinning, for the simple reason that the intermediate value `(+ b c)` did not occur in the original.

The method of inference Thinner uses to discover thinning examples is fast, but not powerful; it sometimes misses thinning examples that seem reasonably clear to a human programmer. Thinner uses the method of kernels, which is only an approximate solution to the trace grammar thinning problem, and it does not compute an exhaustive kernel set. So there are cases in which Thinner will identify a thinning example but not be able to remove all instances of that example from the grammar.

One final limitation of Thinner involves the way it searches for thinning examples: depth-first iterative deepening, stopping at the first violation of the PLC it sees. Thinner may start materializing the constant bases cases of a recursion. (It does so in the Fibonacci example, after the first few thinnings.) Once such a function appears in the grammar, Thinner will never seek any deeper. It will always find the next base case at the same depth of derivation. This can mask other more useful optimizations that might be found if Thinner searched more deeply.

Table I. Experiments with Thinner

Original	Optimized	Time
(defun cse3 (a b c) (f (+ a b) c (* c (+ a b)))) (defun f (x y z) (+ (* y x) z))	(defun cse3 (a b c) (let ((d (* c (+ a b)))) (+ d d))) (defun f (x y z) (+ (* y x) z))	.18s
(defun dce2 (x y) (let ((a (cons 1 x)) (b (cons 0 inil))) (let ((c (append a b)) (d (append y a))) (if (equal c d) (length a) (length b)))))	(defun dce2 (x y) 1)	.08s
(defun sink (a b) (let ((x (* a a))) (if (> a b) (* x b) (* a b))))	(defun sink (a b) (* (if (> a b) (* a a) a) b))	.01s
(defun f (a b c) (if (equal a b) c (+ (if (< b c) a (* a a)) (f (- a 1) b c))))	(defun f (a b c) (if (equal a b) c (multiple-value-bind (v0 v1) (if (< b c) (values a (f1 b c (- a 1))) (values (* a a) (f2 b c (- a 1)))) (+ v0 v1)))) (defun f1 (b c a) (if (equal a b) c (+ a (f1 b c (- a 1))))) (defun f2 (b c a) (if (equal a b) c (+ (* a a) (f2 b c (- a 1)))))	1.9s

5. CONCLUSION

These are some of the open problems in grammatical reformulation:

—The CFG-thinning problem: what other methods for thinning CFGs exist? How do they compare with one another? Can they be lifted into the domain of graph grammars, and so applied to program optimization?

—The problem of finding kernels: how can one find a useful set of kernels quickly? Is it necessary to represent each one explicitly, or is there some more compact representation that would serve?

—The inference problem: given a trace grammar, identify as many thinnings as possible, as quickly as possible. Thinner's technique, relational constraint, produces good results, but more inferential power would be welcome. For example, trace grammars often have a recursive structure that strongly hints at a path for

Table II. More Experiments with Thinner

Original	Optimized	Time

(defun f (i start)
  (values (sum i start)
  (sumsq i start)))
(defun sum (i sofar)
  (if (equal i 0)
  sofar
  (sum (- i 1) (+ sofar i))))
(defun sumsq (i sofar)
  (if (equal i 0)
  sofar
  (sumsq (- i 1)
  (+ sofar (* i i)))))
	(defun f (i start)
	  (if (equal i 0)
	  (values start start)
	  (let ((m (- i 1)))
	  (if (equal m 0)
	  (let ((n (+ start i)))
	  (values n n))
	  (h (+ start (* i i))
	  (+ (+ start i) m)
	  m
	  (- m 1))))))
	(defun h (a b c d)
	  (if (equal d 0)
	  (values b (+ a 1))
	  (h (+ a (* c c))
	  (+ b d)
	  d
	  (- d 1))))
		8.1s

(defun fib (a)
  (if (< a 2)
  a
  (+ (fib (- a 1)) (fib (- a 2)))))
	(defun fib (a)
	  (if (< a 2)
	  a
	  (multiple-value-bind (x y)
	  (h (- a 1))
	  x)))
	(defun h (b)
	  (if (< b 2)
	  (values b b)
	  (multiple-value-bind (x y)
	  (h (- b 1))
	  (values (+ x y) x))))
		1.5s

an inductive proof, but Thinner performs no induction.

Thinner is a proof of concept, not a practical optimizer; there are no TL programs outside the Thinner project. And we do not propose to build a practical optimizer that painfully rediscovers common compiler optimizations over and over! The canon of compiler optimizations has its place: these are precisely the optimizations that experience tells us are easy to identify, and occur often enough to be worth looking for. It is illuminating to see that Thinner can perform loop invariant removal, but grammar thinning is a very big hammer for little nails like this.

Thinner works by searching blindly for a thinning example, then applying the method of kernels. This makes for a nice demonstration of the method, but it is not a practical approach to program optimization. The most important open problem in grammar thinning is the control problem: when should an optimizer apply grammar thinning? It should be possible to build an optimizer that chooses grammar thinning only when its unique power and generality are needed.

ACKNOWLEDGMENTS

I am grateful for the advice and encouragement of Devika Subramanian, who proposed the problem treated in this article. Thanks to Sam Kamin and to the anonymous referees for their valuable suggestions.

REFERENCES

ARVIND AND NIKHIL, R. 1987. Executing a program on the MIT tagged-token dataflow architecture. In *PARLE: Parallel Architectures and Languages Europe.* Vol. 2. Springer-Verlag, Berlin, 1–29.

BACKUS, J. 1985. From function level semantics to program transformation and optimization. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, Eds. Lecture Notes in Computer Science, vol. 185. Springer-Verlag, Berlin, 60–91.

BUNKE, H AND HALLER, B. 1990. A parser for context free plex grammars. In *WG '89 Graph-Theoretic Concepts in Computer Science: Proceedings of the 15th International Workshop*, M. Nagl, Ed. Lecture Notes in Computer Science, vol. 411. Springer-Verlag, Berlin, 136–150.

BURSTALL. R. AND DARLINGTON. J. 1977. A transformation system for developing recursive programs. *J. ACM 24,* 1 (Jan.), 44–67.

CHOMSKY N. 1962. Context-free grammars and pushdown storage. In *Q. Prog. Rep. 65*, Res Lab. Elect., MIT, Cambridge, Mass., 187–194.

COCKE. J. AND SCHWARTZ. J. T. 1970. Programming languages and their compilers; preliminary notes. Courant Inst. of Mathematical Sciences, New York Univ., New York.

COHEN. N. H. 1983. Eliminating redundant recursive calls. *ACM Trans. Program. Lang. Syst. 5,* 3 (July), 265–299.

CONSEL. C. AND DANVY. O. 1993. Tutorial notes on partial evaluation. In *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, New York, 493–501.

DOWNEY. P. J , SETHI, R., AND TARJAN. R. E. 1980. Variations on the common subexpression problem. *J. ACM 27,* 4 (Oct.), 758–771.

FEDER. J. 1971. Plex languages. *Inf. Sci. 3*, 225–241.

FEYNMAN. R. P., LEIGHTON. R. B., AND SANDS. M. 1964. *The Feynman Lectures on Physics.* Vol. 2. Addison-Wesley, Reading, Mass.

GAREY. M. R. AND JOHNSON. D. S. 1979. *Computers and Intractability.* W. H. Freeman, New York.

GUNTER, C. A. AND SCOTT. D. S. 1990. Semantic domains. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. The MIT Press/Elsevier, Cambridge, Mass., chap. 12.

HALL. R. J. 1990. Program improvement by automatic redistribution of intermediate results. Tech. Rep. AI-TR 1251, MIT AI Lab, Cambridge, Mass. Dec.

INDERMARK. K. AND KLAEREN. H. 1987. Compiling fibonacci-like recursion. *SIGPLAN Not. 22,* 6 (June), 101–107.

JANTZEN. J. AND PETERSEN. H. 1994. Cancellation in context-free languages: Enrichment by reduction. *Theor. Comput. Sci. 127*, 149–170.

JOHNSON. J. H. 1986. Rational equivalence relations. *Theor. Comput. Sci. 47,* 1, 39–60.

JONES. J. AND MATIJASEVIC. Y. 1984. Register machine proof of the theorem on exponential diophantine representation of enumerable sets. *J. Symbol. Logic 49,* 3 (Sept), 818–829.

KIEBURTZ R. B AND SHULTIS, J 1981. Transformations of FP program schemes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* ACM, New York.

KNUTH. D. E., MORRIS. J H., AND PRATT. V R 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2 (June), 323–350.

MOREL. E. AND RENVOISE. C 1979. Global optimization by supression of partial redundancies. *Commun. ACM 22,* 2 (Feb.), 96–103.

PERLIN, M. 1989. Call-graph caching: Transforming programs into networks. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence.* IJCAII, Morristown, N.J., 122–128

ROHL, J. S. 1989 Recursive and iterative functions for generating Fibonacci numbers. Tech Rep. TR 89-1017, Cornell Univ , Ithaca, N.Y. June.

ROSEN B K , WEGMAN. M. N , AND ZADECK F K. 1988. Global value numbers and redundant computation. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 12–27

ROZENBERG. G. 1987. An introduction to the NLC way of rewriting graphs. In *Graph Grammars and Their Application to Computer Science*, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, Eds. Lecture Notes in Computer Science, vol. 291. Springer-Verlag, Berlin.

SALOMAA, A 1973. *Formal Languages.* Academic Press, New York.

SØRENSEN, M H , GLÜCK, R., AND JONES, N. D 1994. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP '94: 5th European Symposium on Programming* (Edinburgh, U.K.). Lecture Notes in Computer Science, vol. 788. Springer-Verlag, Berlin, 485–500.

STEELE G. 1990. *Common Lisp: The Language*, 2nd ed. Digital Press, Bedford, Mass.

SUBRAMANIAN, D 1989. A theory of justified reformulations Ph D thesis, Stanford Univ , Stanford, Calif.

SUBRAMANIAN. D AND GENESERETH. M. R. 1987. The relevance of irrelevance. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence.* IJCAII, Morristown, N.J.

TURCHIN, V F. 1986. The concept of a supercompiler *ACM Trans. Program. Lang. Syst. 8,* 3 (July), 292–325.

TURCHIN, V F 1980. The use of metasystem transition in theorem proving and program optimization. In *Automata, Languages and Programming, the 7th Colloquium.* Lecture Notes in Computer Science, vol. 85. Springer-Verlag, Berlin, 645–657.

TURCHIN, V. F , NIRENBERG, R. M , AND TURCHIN. D V 1982. Experiments with a supercompiler. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming.* ACM, New York, 47–55.

WEBBER, A. B 1993. Principled optimization of functional programs. Ph.D. thesis, Cornell Univ., Ithaca, N.Y.

WEBBER, A. B. 1992. Relational constraint: A fast semantic analysis technique. Tech. Rep. TR 92-1319, Cornell Univ., Ithaca, N.Y. Dec.

Received March 1994; revised December 1994; accepted January 1995