Thinning Context-Free Languages

Adam Brooks Webber*

TR 92-1318
December 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Supported by NSF grant IRI-8902721.
Thinning Context-Free Languages

Adam Brooks Webber*
Computer Science Department
Cornell University
Ithaca, NY 14853
webber@cs.cornell.edu

August 1992

Abstract

A thinning is a kind of transformation suggested by a practical problem in program optimization. We're given a thinning example \( \delta \), which is a string with one or more symbols marked. To thin a language with respect to this \( \delta \) is to edit each word to reflect the observation that the marked symbols of \( \delta \) are unnecessary and should be omitted. In this paper we'll explore thinnings of context-free languages.

When \( \delta \) is a fully-terminal string we'll define a language \( L_\delta(G) \) formed by carrying out a thinning procedure on each word in \( L(G) \). For a large class of thinning examples—the class for which the language of strings correctly marked for thinning is regular—we will demonstrate that \( L_\delta(G) \) is a context-free language and that there is an effective procedure for generating a grammar \( H \) for which \( L(H) = L_\delta(G) \).

When \( \delta \) may include non-terminals the problem is more difficult to formalize: we'll define a language \( L_\delta(G) \) formed by carrying out a thinning procedure on each parse tree generated by \( G \). This yields an extension of the fully-terminal problem which is unsolved and seems very hard. An easier problem follows from defining a language \( L_\Delta(G) \) formed by thinning the parse trees of \( G \) with respect to a set of thinning-tree examples instead of a single thinning-string example. We will develop an effective procedure for generating a grammar \( H \) for which \( L(H) = L_\Delta(G) \). By choosing an appropriate thinning set \( \Delta \), we can use this method to get an approximate solution to the general problem.

*Supported by NSF grant IRI-8902721. Copyright ©1992 by Adam Brooks Webber. All rights reserved.
We will conclude with an example showing how the \( L_\Delta \) method of context-free grammar transformation can be adapted for use on trace grammars, a type of graph grammar used in the Thinner project to represent functional programs. There are several open problems in the thinning of context-free grammars which are all the more interesting because of this potential application to program optimization.

1 The Idea of Thinning

The Thinner project is an attempt to formalize a broad and intuitive principle of optimization, and to construct a fully-automatic optimizer that finds and corrects as many violations of this principle as it can [Web92]. The principle in question is the Principle of Least Computation\(^1\) or PLC: it’s the rather obvious principle that programs shouldn’t make any unnecessary computations. Consider the computation of two values shown in Figure 1.

![Figure 1: A violation of the PLC](image)

This fragment violates the PLC since there’s no need to compute both \( f(x - 2) \) and \( f(x - 1 - 1) \). A transformation that repairs a violation of the PLC is called a thinning: it removes some computations from a program without adding or changing anything else.

Thinnings are easy to carry out as long as the program in question has a simple linear structure. But consider the following functional program:

```
(defun f (x)
  (if (< x 2)
    x
    (+ (f (- x 1)) (f (- x 2)))))
```

This is the exponential-time implementation of the Fibonacci function encountered by every computer science student. This program certainly violates the PLC, making a large number of unnecessary computations. In fact.

\(^1\)From Feynman’s Principle of Least Action[FLS64].
it violates the PLC with the same redundant computation shown in the example of Figure 1. (To see this, unfold \( f \ (\sim \ x \ 1) \) once in the definition of \( f \)) But what can be done about it? What transformation will prevent the program from making this mistake?

Of course the answer in this case is well known, but it would be nice to have a general method for solving this kind of problem. A thinning example is a kind of cautionary tale: it demonstrates a mistake, a waste of computation, that programs should avoid. Given a program and a thinning example, we want to be able to modify the program so that it never makes the mistake demonstrated in the example.

When I started thinking about this program-thinning problem I decided that an easy line of attack would be to treat an analogous problem involving string languages: a grammar-thinning problem for context-free grammars. It turned out to be quite a rich area (and not a particularly easy line of attack). At least one grammar-thinning method translates well to the program-thinning problem; Section 3 sketches an example of this. Other methods may also carry over to program optimization, and work on this is still under way. But in this paper we will concentrate on the thinning problem as it applies to context-free languages.

2 The Thinning Problem for CFGs

In the context-free grammar version of the thinning problem we’re given a CFG, and a thinning example which is a string \( \delta \) 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 \( \delta \) is a fully terminal string the problem is fairly easy to state rigorously, and we’ll deal with this case first.

2.1 A Fully Terminal Thinning Problem

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

function ThinString\( (x, \delta) \);

\( x \) is a string of terminals;
\( \delta \) is a string with at least one marked symbol;
ThinString\((x, \delta)\) is a string of terminals (a subsequence of \(x\));
\begin{verbatim}
begin
  while there is an instance\(^2\) of \(\delta\) in \(x\)
  begin
    find the leftmost instance of \(\delta\) in \(x\);
    delete from \(x\) those symbols that match marked symbols in \(\delta\)
  end;
  return \(x\)
end;
\end{verbatim}

For example, if \(x = aabc\) and \(\delta = ab\) (where underscored indicates a marked character), ThinString deletes symbols from \(x\) to end up with the string \(bc\): first it finds the instance of \(ab\) in \(aabc\) and deletes that \(a\) to get \(abc\); then it finds the instance of \(ab\) in \(abc\) and deletes that \(a\) to get \(bc\). The reader may wish to try an example or two. Try thinning the string \(xyxyxyxyxy\) using \(\delta = zxy\). (Remember to thin the leftmost instance of \(\delta\) each time.) Or try thinning \(abxaabxaabxaabxaabxa\) using \(\delta = abzab\). 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 \(\delta\). That is,
\[
L_\delta(G) = \{ w \mid w = \text{ThinString}(v, \delta) \text{ for some } v \in L(G) \}
\]

This is like learning the lesson of \(\delta\) 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 \(L(H) = L_\delta(G)\). But this is jumping ahead a bit. There are really two questions here:

1. Is \(L_\delta(G)\) a context-free language for any CFG \(G\) and any thinning example \(\delta\)?

2. Can we give an algorithm that takes a CFG \(G\) and a thinning example \(\delta\) and produces a new CFG \(H\) for which \(L(H) = L_\delta(G)\)?

The answer to these questions is unknown. However, there is a special case (actually, a common case) for which both questions can be answered affirmatively. This case allows a general CFG \(G\) but restricts the thinning example \(\delta\) as described below.

Define a language \(M(\delta)\) of strings “correctly marked for thinning” as follows: a string \(z\) is in \(M(\delta)\) iff exactly those characters deleted when \(z\) is

\(^2\)By “instance” I mean substring or, as some authors have it, factor.
editing by \( \delta \) are marked in \( z \). So \( M(ab) \) includes \( abc \) and \( aabc \) but not \( abc \) (because a character deleted during editing by \( ab \) is not marked) and not \( a \epsilon c \) (because a marked character is not deleted). An immediate observation is that \( M(\delta) \) is context-free.

**Theorem 1** For any \( \delta \), \( M(\delta) \) is a context-free language.

**Proof:** by construction of a pushdown automaton \( P \) which accepts \( M(\delta) \) by empty stack. In the first phase of its operation, \( P \) carries out the following step repeatedly: if there is a \( \delta \) on top of the stack, one in which all the symbols to be deleted are marked, delete those symbols; if there is no \( \delta \) on top of the stack, either shift one symbol from the input string to the top of the stack and continue the first phase, or move on to the second phase. (Note that if there is a \( \delta \) in which a symbol to be deleted is not marked, no operation is defined and \( P \) hangs.) In the second phase, it is assumed that the input string is exhausted, and \( P \) accepts no further input symbols. As long as the top symbol on the stack is unmarked, \( P \) pops that symbol.

Think of the contents of the stack concatenated with the unused input as a string on which \( P \) is working in the first phase. This string changes only when \( P \) edits an instance of \( \delta \), so \( P \) generates a sequence of strings starting with the initial input and ending with the contents of the stack at the end of the first phase. A simple inductive proof shows that this is the same sequence of strings generated by ThinString: in effect, the first phase of \( P \) edits the input string by \( \delta \) while transferring it to the stack.

The first phase of \( P \) completes if and only if all characters deleted during editing by \( \delta \) were marked. The second phase empties the stack if and only if all marked characters were deleted during editing. So \( P \) accepts \( M(\delta) \). \( \square \)

It often happens that \( M(\delta) \) is not only context-free but regular. This is the kind of \( \delta \) for which we can give a solution to the grammar-thinning problem.

**Theorem 2** If \( M(\delta) \) is a regular language then \( L_\delta(G) \) is context-free for any CFG \( G \).

**Proof:** Given a pushdown automaton \( P \) accepting \( L(G) \) by final state, and given a finite-state automaton \( F \) accepting \( M(\delta) \), construct a pushdown automaton \( P_\delta \) as follows. \( P_\delta \) can augment its input string nondeterministically: whenever it is about to scan an input symbol, it may decide to leave the
real next input symbol for later and supply any marked input symbol instead. Using this augmented input string, \( P_\delta \) simulates both \( P \) (ignoring the difference between marked and unmarked symbols) and \( F \) (respecting the difference between marked and unmarked symbols). The accepting states of \( P_\delta \) are those in which both \( P \) and \( F \) are in accepting states.

An input string \( w \) is accepted by \( P_\delta \) if and only if there is some way to augment \( w \) with marked symbols yielding a string \( w' \) which is in \( L(G) \) (ignoring marking) and in \( M(\delta) \) (respecting marking). This is true if and only if there is some \( w' \in L(G) \) that yields \( w \) when thinned by \( \delta \). So \( P_\delta \) accepts \( w \) if and only if \( w \in L_\delta(G) \). □

Of course, we don’t just want to know that the thinned language is context-free; we want to be able to construct a grammar \( H \) for which \( L(H) = L_\delta(G) \). (Remember, our motivation involves the optimization of computer programs; a proof that a thinned program exists may not be that useful.) Of course we could use the roundabout method suggested by the proof: construct a pushdown automaton \( P \) for \( G \), using it and the finite-state automaton \( F \) for \( M(\delta) \) to construct \( P_\delta \) as described above, and then using the classical construction (see, for example, [HU79]) to make a grammar \( H \) that generates the language accepted by \( P_\delta \). The resulting grammar would be complicated, though, and it would bear little intuitive relation to the original one. Fortunately, there is a more direct way to construct \( H \), as suggested by the following alternative proof of Theorem 2.

Alternate Proof of Theorem 2: Suppose \( G \) is a CFG and \( \delta \) is a thinning example for which \( M(\delta) \) is a regular language. If \( L \) is a language without any marked symbols, define \( S(L) \) to be the same language but allowing all patterns of marked and unmarked symbols. That is, \( S(L) \) is a substitution applied to \( L \), which maps each terminal symbol \( x \) to either \( x \) or \( \overline{x} \). Clearly \( S(L(G)) \) is context-free. Since \( M(\delta) \) is regular, the intersection of \( S(L(G)) \) 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 I 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. This is treated in, for example, [Sal73]. The general idea is to specialize each non-terminal \( X \) in
the grammar into a variety of non-terminals \(X_{i,j}\), where \(i\) and \(j\) are states of the FSM for \(M(\delta)\). \(X_{i,j}\) derives a string \(w\) if and only if \(X\) derives \(w\) and \(w\) takes the FSM from state \(i\) to state \(j\). In practice, many of the \(X_{i,j}\)'s are unnecessary: they either derive no fully-terminal string or are not used in any derivation starting from \(S_{s,b}\) (where \(S\) is the start symbol, \(s\) is the start state and \(b\) is an accepting state). My implementation avoids generating these unnecessary non-terminals. It also simplifies the resulting grammar a bit, getting rid of self-productions, duplicate non-terminals (i.e. those with identical productions), and non-terminals with single productions. Figure 2 shows a grammar thinning performed by this implementation.

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow aA \mid a \mid c \\
B & \rightarrow bB \mid b
\end{align*}
\]

**Initial grammar \(G\)**

\[
\begin{align*}
S & \rightarrow S_{0,0} \mid S_{0,1} \\
S_{0,0} & \rightarrow A_{0,0}B_{0,0} \\
S_{0,1} & \rightarrow A_{0,1}B_{1,1} \\
A_{0,0} & \rightarrow aA_{1,0} \mid c \\
A_{0,1} & \rightarrow aA_{1,1} \mid a \\
A_{1,0} & \rightarrow aA_{1,0} \mid c \\
A_{1,1} & \rightarrow aA_{1,1} \mid a \\
B_{0,0} & \rightarrow bB_{0,0} \mid b \\
B_{1,1} & \rightarrow B_{1,1} \mid \epsilon
\end{align*}
\]

**FSM for \(M(\delta)\), \(\delta = ab\). Start state is 0 and both states accept.**

\[
\begin{align*}
S & \rightarrow A_{x,0}B_{0,0} \mid A_{x,1} \\
A_{x,0} & \rightarrow aA_{x,0} \mid c \\
A_{x,1} & \rightarrow aA_{x,1} \mid a \\
B_{0,0} & \rightarrow bB_{0,0} \mid b
\end{align*}
\]

**Final thinned grammar**

**Figure 2: Construction of a grammar for \(L_\delta(G)\)**

Since we have a solution to the fully-terminal thinning problem when the thinning example \(\delta\) yields a regular \(M(\delta)\), it makes sense to ask whether there is a more direct characterization: for what kinds of \(\delta\) is \(M(\delta)\) a regular language? Figure 3 shows a selection of thinning examples \(\delta\), categorized according to whether \(M(\delta)\) is regular. (Remember, \(M(\delta)\) is always context-free.) Note that \(M(\delta)\) is regular for many thinning examples: it's regular whenever \(\delta\) has exactly one marked symbol, and it's regular even for some
fairly tricky $\delta$’s like $\delta = xyz$ and $\delta = abzab$ (which were used above in an exercise for the reader).

<table>
<thead>
<tr>
<th>$M(\delta)$ regular</th>
<th>$M(\delta)$ not regular</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ab$</td>
<td>$ab$</td>
</tr>
<tr>
<td>$xyxy$</td>
<td>$xzyyz$</td>
</tr>
<tr>
<td>$aaa$</td>
<td></td>
</tr>
<tr>
<td>$abzab$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Some thinning examples classified

All thinning examples of which I am aware fit a simple pattern:

Conjecture 3 If $M(\delta)$ is non-regular then thinning $\delta$ yields some string $\beta$ for which there are non-empty strings $\alpha$ and $\gamma$ such that $\delta = \alpha \beta \gamma$ (and so $\alpha^n \beta \gamma^n$ thins to $\beta$).

The implication doesn’t work the other way, as shown by the case $\delta = aaaa$. I believe that a strengthened form of this condition is both necessary and sufficient, but I have not yet been able to prove it.

2.2 Thinning With Non-terminals

The trouble with the terminal solution presented above is that it doesn’t seem to apply to the example that motivated this study, the Fibonacci optimization. One of the pieces thinned away in that example is a function call—in effect, a non-terminal. We could extend the terminal solution to make it treat some non-terminals as terminals, but this would only work if the thinned non-terminals never participate in an instance of $\delta$. The Fibonacci example doesn’t appear to fit this case: the thinned non-terminal is a recursive call. We need a different approach.

Suppose $\delta$ contains non-terminals. How can we edit a language to reflect the lesson of this $\delta$? It seems natural to extend our original string thinning function to parse trees. Define an instance of $\delta$ in a tree $T$ to be a string of nodes of $T$ that matches $\delta$ and occurs on some search frontier in $T$.

function ThinTree($T, \delta$);

$T$ is a parse tree;
$\delta$ is a string with at least one marked symbol;
ThinTree($T, \delta$) is a tree, a pruned version of $T$;
begin
\textbf{while} there is an instance of $\delta$ in $T$
begin
find the lexically first instance of $\delta$ in $T$;
delete from $T$ those subtrees that match marked symbols in $\delta$
end;
return $T$
end;

This is an extension of ThinString in the sense that if $\delta$ is fully terminal, it makes no difference whether we thin the fringe of a parse tree using ThinString, or thin the parse tree using ThinTree and then take the fringe: 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 $\delta$, then reading off the remaining terminal string. That is,

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

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

It may be a meaningful definition, but it seems to lead nowhere. We made progress in the fully-terminal case by insisting that $M(\delta)$ be regular; in the non-terminal case, since we’re not editing strings, the question of whether the string language $M(\delta)$ is regular seems irrelevant. To make progress on this problem, we’ll define a weaker thinning procedure: one that can be used to approximate ThinTree, but is computationally less demanding.

\textbf{procedure} SetThinTree($T, \Delta$);
$T$ is a parse tree which is \textit{edited in place};
$\Delta$ is an ordered set of trees, each with at least one marked node;
begin
\textbf{while} some member of $\Delta$ is a prefix of $T$
begin
$Q := \text{the first member of } \Delta \text{ which is a prefix of } T$;
delete from $T$ those subtrees that match marked nodes in $Q$
end
end;
function WeakThinTree($T$, $\Delta$);  
$T$ is a parse tree;  
$\Delta$ is an ordered set of trees, each with at least one marked node;  
WeakThinTree($T$, $\Delta$) is a tree, a pruned version of $T$;  
begin  
for each subtree $Q$ of $T$ in a topological order  
SetThinTree($Q$, $\Delta$);  
return $T$  
end;

Using this procedure, we can define the thinned language $L_\Delta(G)$ as before:

$L_\Delta(G) = \{ w \mid w$ is the fringe of WeakThinTree($T$, $\Delta$) for some $T$ generated by $G$ $\}$

WeakThinTree is certainly weaker than ThinTree. In particular, it is no longer equivalent to ThinString in the fully-terminal case. This is because any particular thinning set $\Delta$ has a fixed maximum tree height; it is easy to construct a grammar in which the distance to the common ancestor of some instance of $\delta$ is unbounded. However, there’s one good thing about weak tree thinning: we can solve the grammar thinning problem with respect to this procedure.

**Theorem 4** $L_\Delta(G)$ is context free for any thinning set $\Delta$ and any CFG $G$.

**Proof:** by construction of a CFG $H$ for which $L(H) = L_\Delta(G)$. We begin by showing that the following procedure carries out weak tree thinning and derivation simultaneously; then we'll construct a CFG that mimics the new procedure.

Define a $\Delta$-deep tree as one in which any non-terminal which is not expanded has distance exactly $n$ from the root, where $n$ is the maximum height of any tree in $\Delta$. (Note that a $\Delta$-deep tree may have paths of length $< n$ as long as their end at terminals.)

**procedure** ExtendThinning($R$, $G$, $\Delta$);  
$R$ is a parse tree which is edited in place;  
$G$ is a context-free grammar;  
$\Delta$ is an ordered set of trees, each with at least one marked node;  
begin
1 if $R$ is less than $\Delta$-deep, extend it using grammar $G$;
2 if some member of $\Delta$ is a prefix of $R$
3    then
4        begin
5            $Q :=$ the first member of $\Delta$ which is a prefix of $R$;
6            delete from $R$ those subtrees that match marked nodes in $Q$;
7            ExtendThinning($R, G, \Delta$)
8        end
9    else
10       for each non-terminal child $Q$ of $R$ in order
11          ExtendThinning($Q, G, \Delta$)
end;

The idea is to apply this procedure using the start symbol of $G$ as the tree to extend: $\text{ExtendThinning}(S, G, \Delta)$. With various “extensions by grammar $G$” at line 1 this will generate various trees. The recursive call at line 7 thins the developing tree iteratively, just as $\text{WeakThinTree}$ does when it calls $\text{SetThinTree}$. The recursive call at line 11 visits each subtree in a topological order, just as $\text{WeakThinTree}$ does. (As long as it's topological, the actual order is irrelevant, since the subtrees are thinned independently.) The only thing different about $\text{ExtendThinning}$ is that it derives just enough of each subtree to decide how to thin it, leaving the rest until the descendant subtrees are investigated. So the language of trees generated by $\text{ExtendThinning}(S, G, \Delta)$ is the same as the language of trees generated from $G$ in the normal way and then thinned by $\text{WeakThinTree}$ using the thinning set $\Delta$.

Observe that $\text{ExtendThinning}(S, G, \Delta)$ never results in an $\text{ExtendThinning}$ call with a tree that is more than $\Delta$-deep. Also, the maximum degree in each tree is bounded by the maximum length of the right-hand side of any production in $G$. It follows that even though $\text{ExtendThinning}$ may develop an arbitrarily-large thinned parse tree, the portion of it that it examines at any stage is bounded, fixed by $G$ and $\Delta$. Thus there are only finitely-many different parameters $R$ on which $\text{ExtendThinning}$ will call itself.

Now we construct a CFG $H$ that mimics the behavior of $\text{ExtendThinning}$. The terminal alphabet of $H$ will be the same as the terminal alphabet of $G$, but the non-terminal alphabet of $H$ will consist of a start symbol $Z_S$, where $S$ is the start symbol of $G$, and various other symbols denoted $Z_R$, where $R$ is one of the finitely-many different tree parameters on which $\text{ExtendThinning}$ will call itself. For each non-terminal symbol $Z_R$ in $H$ the
productions are determined as follows. Let \( X \) be the set of \( \Delta \)-deep expansions of \( R \) using grammar \( G \); this will be just \( \{ R \} \) if \( R \) is already \( \Delta \)-deep. (\( X \) corresponds to the set of possible extensions generated at line 1.) For each \( S \in X \) with a prefix in \( \Delta \) there is a production \( Z_R \rightarrow Z_Q \), where \( Q \) is the result of thinning \( S \) using the first prefix of \( S \) in \( \Delta \). (\( Q \) corresponds to the thinning computed at lines 5 and 6.) For each \( S \in X \) with no prefix in \( \Delta \), let \( Q_i \) be the \( i \)th son of the root of \( S \); there is a production \( Z_R \rightarrow P_1 \cdots P_n \), where \( P_i \) is either \( Q_i \) if \( Q_i \) is a terminal node, or \( ZQ_i \) if \( Q_i \) is not a terminal node. (This corresponds to recursively visiting each non-terminal child at lines 10 and 11.)

By construction, there is a one-to-one correspondence between derivations in \( H \) and traces of ExtendThinning\( (S, G, \Delta) \). Since we know ExtendThinning generates \( L_\Delta(G) \), we can conclude that \( L(H) = L_\Delta(G) \). \( \Box \)

This proof suggests a method for transforming grammars—a method I have implemented. Given a thinning example \( \delta \) and a grammar \( G \), I generate a thinning set \( \Delta \) of all the derivations in \( G \) that generate \( \delta \) of height less than or equal to some small constant \( d \). Then I generate the grammar \( H \) using a variation of the method above. The important thing about the method above is that it constructs non-terminals which stand for latent parse trees, thus holding pieces of the parse tree together until it is known how to thin them. My implementation uses this same technique, but does not necessarily generate \( \Delta \)-deep trees, as ExtendThinning does at line 1; instead, it extends a tree only until the next thinning step is determined. Figure 4 shows a grammar thinning as generated by this implementation.

3 The Motivating Problem Revisited

We have derived two methods for transforming a context-free grammar given a thinning example. Work on adapting these grammar-transformation methods for use on computer programs is now under way as part of the Thinner project. It’s beyond the scope of this paper to treat program-thinning methods in detail; but, since program optimization was our motivation for studying the thinning of context-free languages, we’ll conclude by sketching the program-thinning method that derives from the \( L_\Delta \) method of thinning context-free grammars.

Thinner is an optimizer for first-order, purely functional programs. It
uses a special representation for the programs it tries to optimize. The basis of this representation is the trace graph, a directed acyclic graph structure that describes a single path of computation, a single mapping from inputs to outputs by way of intermediate values, without any loops or conditionals. The vertices of a trace graph represent primitive functions and the arcs represent intermediate values. Of course some of the computations in the course of a normal program's execution will be predicates: comparisons, tests, or other primitives, each with a single true-or-false output. These predicates determine which one of the many paths of execution implicit in a program is actually followed. Any predicate evaluated in the course of a particular path of execution will appear in the trace graph for that path, but the output of that predicate will be fixed by the graph structure. A trace graph is like a partial evaluation of a program—a partial evaluation with respect to a fixed control path, as in the work of Perlin [Per89]. It has an execution only for those inputs that result in the proper predicate values.

The trace graph isn't powerful enough by itself to represent a program: a program may have infinitely many possible paths of execution, so we need some way to generate potentially infinite sets of trace graphs. Thinner uses a kind of graph grammar, the trace grammar, for this purpose. It is a spe-
cialization of the directed, node-label controlled (DNLC) graph grammars studied by Jannsens and Rozenberg [Ro287]. It works this way: a trace graph may contain vertices whose functions are non-terminals, and each non-terminal appears on the left-hand side of some production in the trace grammar. The right-hand side is a “daughter” trace graph whose input and output arities and types match those of the “mother” vertex. The host trace graph is transformed into a new graph by vertex replacement: the mother vertex is removed and the daughter graph is inserted in its place. The set of fully-terminal trace graphs that can be generated from the start graph by repetitions of this process is the set generated by the trace grammar.

This representation is described in more detail in [Web92]; for now, the example of Figure 5 will have to serve. This is the trace grammar generated by Thinner for the Fibonacci function we saw earlier. Note that if the recursive vertex highlighted in the diagram is expanded using the second production, the resulting graph contains the thinning example of Figure 1 as a subgraph.

\[
\{ (f \rightarrow 2), (f \rightarrow 1) \}
\]

Figure 5: Initial trace grammar for \( f \), highlighted for unfolding

We could just make an unfolded version of \( f \) and fix the thinning example where it occurs in a production, but the grammar would still generate the same violation after further unfolding. (In effect, we would still have an exponential-time program, although the exponent would be reduced.) Instead, we identify all the possible derivations that lead from a non-terminal to our thinning example—this corresponds to computing a thinning-set \( \Delta \) from a thinning example \( \delta \). Figure 6 shows the only path of derivation that leads to the thinning example (we show it as a derivation path rather than a parse tree because of the difficulty of illustrating a parse trees whose nodes are graphs). Graph \( C \) in this figure is the thinning example. Graph \( B \) is the minimal predecessor of \( C \): unfolding the highlighted nonterminal in \( B \)
results in a graph that contains $C$, and everything else in $B$ occurs in $C$. Graph $A$ is just the non-terminal $f$, which is the minimal predecessor of $B$.

```
A  B  C
```

Figure 6: Derivation path for the thinning example

Subgraph $B$ is the troublemaker: it may or may not unfold into something that includes the thinning example. It is, in effect, the fringe of a parse tree for which the next thinning step is not yet determined. So we create a new nonterminal $f2$ which, like the nonterminals $Z_R$ in the $L_\Delta(G)$ construction, holds the derivation together until the next thinning step becomes clear. Productions for this new non-terminal are built by starting with $B$ and unfolding the marked non-terminal. Using this new nonterminal $f2$ in place of $B$ yields the grammar shown in Figure 7.

```
(f \rightarrow \{<2 1\},(f \rightarrow \{<2 f\},(f2 \rightarrow \{<2 f\},(f2 \rightarrow \})
```

Figure 7: After substituting a new nonterminal $f2$ for subgraph $B$

The new grammar has this useful property: the only way it can generate the thinning example is in the second production of $f2$. Thinning that production now corrects the problem completely, and the resulting grammar is shown in Figure 8.

Note that additional thinning has been performed on the first production of $f2$. (Since its input $x$ is an integer not less than 2, if $x-1 < 2$ it follows that $x = 2$. Thinner recognizes this and substitutes the constant 0 for $f(2-2)$.)
After thinning, the grammar corresponds to the linear-time\(^3\) program:

\[
\begin{align*}
\text{(defun } f \text{ (x)} & \text{)} \\
\text{ (declare (integer x))} & \\
\text{ (if (< x 2) } & \\
\text{ x} & \\
\text{ (multiple-value-bind (a b) (f2 x) } & \\
\text{ (+ a b)))))} & \\
\text{(defun } f2 \text{ (x)} & \text{)} \\
\text{ (declare (integer x))} & \\
\text{ (let ((y (- x 1)))} & \\
\text{ (if (< y 2) } & \\
\text{ (values y 0)} & \\
\text{ (multiple-value-bind (a b) (f2 y) } & \\
\text{ (values (+ a b) a)))))}
\end{align*}
\]

4 Conclusion

The thinning problem for context-free grammars can be formalized in different ways. When \(\delta\) is a fully-terminal string the formalization was straightforward: we defined a language \(L_\delta(G)\) formed by carrying out a thinning procedure ThinString on each word in \(L(G)\). For a large class of thinning examples—the class for which the language of strings correctly marked for thinning is regular—we demonstrated that \(L_\delta(G)\) is a context-free language and that there is an effective procedure for generating a grammar \(H\) for

\(^3\)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: see [Rob89].
which $L(H) = L_\delta(G)$. For fully terminal thinning examples outside this class the thinning problem is open. The problem of giving a direct characterization of the "regular" thinning examples is also open.

When $\delta$ may include non-terminals the problem is more difficult to formalize: we defined a language $L_\delta(G)$ formed by carrying out a thinning procedure ThinTree on each parse tree generated by $G$. This yielded an extension of the fully-terminal problem which is unsolved and seems very hard. An easier problem followed from defining a language $L_\Delta(G)$ formed by thinning the parse trees of $G$ with respect to set of thinning-tree examples instead of a single thinning-string example. This problem proved to be solvable, and we gave an effective procedure for generating a grammar $H$ for which $L(H) = L_\Delta(G)$. By choosing an appropriate set $\Delta$, this can be made to approximate a solution to the general problem; but it cannot be an exact solution to the general problem unless all the instances of $\delta$ generated by $G$ have a common ancestor at a fixed maximum distance in the parse tree.

Current work on the Thinner project involves adapting grammar-thinning methods for use on trace grammars, a type of graph grammar for representing functional programs. I expect this to yield an interesting class of program optimizations; it certainly provides a motivation for further work on the thinning of context-free grammars.

5 Acknowledgements

Thanks to Dexter Kozen, Juris Hartmanis and Devika Subramanian for several helpful discussions on this subject.

References


