Principled Optimization of Functional Programs

Adam Brooks Webber
Ph.D Thesis

93-1363
June 1993

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

PRINCIPLED OPTIMIZATION OF FUNCTIONAL PROGRAMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Adam Brooks Webber

August 1993

PRINCIPLED OPTIMIZATION

OF

FUNCTIONAL PROGRAMS

Adam Brooks Webber, Ph.D.

Cornell University 1993

Automatic optimizers for computer programs work with a fixed list of rote transformations, while human programmers can go on to derive new optimizations from broad and intuitive principles if known transformations prove inadequate. This dissertation investigates the possibility that principled optimization can be automated, focusing on a single principle (the principle that programs should not do anything unnecessary) and a single program domain (the domain of purely functional, first-order programs). Three questions are explored: How can the principle be formalized? How can violations of the principle be detected? How can violations be repaired?

The *trace grammar* is a new representation for first-order functional programs. It permits a simple formal statement of the optimizing principle. A trace grammar is a kind of graph grammar. An individual graph represents a path of execution through the program, without any loops or conditionals. The grammar for a

program generates a language of graphs representing the possible paths of execution through that program. Trace grammars provide unique leverage for the twin problems of identifying and repairing violations of the principle.

Detecting violations of the principle is a problem in semantic analysis. A new method of inference, the *relational constraint method*, helps make this tractable. The method treats a trace graph as a system of constraints on a lattice of binary relations, and uses those constraints to develop the strongest relation it can find for each pair of values in the graph.

Repairing violations is not easy: given an example of an unnecessary computation performed by the program, one wants to modify the program so that it never makes that mistake. This *grammar thinning* problem for trace grammars corresponds to an interesting open problem on context-free grammars. An (approximate) solution to this CFG problem yields an optimization technique for trace grammars.

An optimizer called Thinner is the proof-of-concept for these ideas. Using the techniques outlined above, Thinner rediscovers a variety of common compiler optimizations. It also finds other, more exotic transformations, including the well-known Fibonacci reformulation. Thinner demonstrates the potential of the principled approach as a high-powered optimizing tool.

Biographical Sketch

Adam Brooks Webber spent his youth in Princeton, Illinois, where he attended Princeton High School and met his wife, Kelly. He went to Dartmouth College in 1980 intending to study drama, but it didn't turn out that way. He worked instead on the Dartmouth College Timesharing System as a systems programmer, and graduated in 1984 with a B.A. in mathematics.

He worked as a software engineer at True BASIC Inc. in Hanover, New Hampshire for four years. Convinced that he had been a more interesting person when in school, he entered the graduate program in Computer Science at Cornell University in 1988. He received the M.S. in 1991 and the Ph.D. in 1993.

dedicated to

Robert Alfred Webber

1895-1989

Acknowledgements

I would like to thank my advisor, Devika Subramanian. Her encouragement and faith have been unfailing. She suggested the problem solved in this thesis, and she taught me respect for AI. I would also like to thank my other thesis advisors, Miriam Leeser and Keshav Pingali.

The research reported here was supported in part by NSF grant IRI-8902721. Thanks also to General Electric for the fellowship that supported me in my first year.

I was sustained in this work by my parents' belief that I could do it. This is a kind of support one never outgrows.

My wife Kelly worked as a waitperson to make this thesis possible. Greater love hath no one than this. Yet my debt to her is no greater than it was five years ago; it was already infinite, and has been ever since she had the goodness to marry me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Consider the following programs:

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

(defun sink (a)
  (let ((x (* a 10)))
    (if (P a) x 0)))

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

What would an ordinary optimizer do with these programs? Of course it depends on the optimizer, but here's a reasonable guess: common subexpression elimination probably will be performed for the first program, code sinking might be performed for the second, and the third almost certainly will not be optimized at all. One thing that can be said for certain is that the ordinary optimizer's repertoire, whether large or small, is fixed: giving it another hour to think about the problem will not help.

What if we give these programs to a human programmer to optimize? Again, it depends on the individual. Some of us have seen all three of these patterns of optimization so often that we can perform the optimizations without much thought. But even a less experienced programmer—say, someone who has only recently learned their first programming language—will be able to discover optimizations in all three cases *if given enough time.*

This is a challenge to computer science. It's something that people can do better than machines; and unlike, say, many natural language or vision or robotics problems, it doesn't seem like the sort of thing for which evolution has prepared us. Machines should have a home-court advantage on problems like this. In fact, mechanical optimizers are superior to human ones in several respects. They're faster, they're cheaper, and they make fewer mistakes. But in their ability to discover new optimizations when known ones prove inadequate, human programmers are unchallenged.

An important component of this wonderful competence is the ability to derive specific optimizations from high-level, intuitive principles. It seems impossible to explain what's wrong with the three programs above without appealing to a particular intuitive principle. In describing this principle we use words like redundant, wasteful, unnecessary; and we feel no need to explain why it is that redundant, wasteful and unnecessary computations should be avoided. There is no lecture in Computer Science 100 devoted to the subject. Even computer neophytes can be expected to know enough to avoid unnecessary computations: after all, we are at least in part computational engines ourselves, and we seem to make it through the day without getting bogged down in pointless computation.

There may be many specific powers behind human superiority at the task of optimization, but the ability to reason from high-level principles to individual optimizations seems like an important one. There may be many principles of optimization, but the principle that programs should not perform any unnecessary computation seems paramount. I call this principle the Principle of Least Computation or PLC.[1]

1.2 Outline

The capstone of this research is a novel optimizer called Thinner. Thinner optimizes a Lisp-like language of first-order functional programs. It works by detecting and correcting violations of the PLC. It discovers traditional compiler optimizations and other, more exotic transformations, including the reformulation of the Fibonacci function from exponential to linear time. Like a human programmer, it can find more optimizations the more time you give it.

To build Thinner I needed answers to three big questions about the Principle of Least Computation:

1. How can the principle be formalized?

2. How can violations of the principle be detected?

3. How can violations of the principle be repaired?

[1] From Feynman's Principle of Least Action [FLS64].

How can the principle be formalized? To answer this first question I developed a new representation for first-order functional programs, the *trace grammar.* A trace grammar is a node-replacement graph grammar. The grammar corresponding to a program generates a language of fully terminal *trace graphs*, each of which describes a single path of execution through the program. The trace grammar representation supports a simple, rigorous expression of the principle. A simple transformation of trace graphs called a *thinning* yields a partial ordering on graphs: one graph can be "thinner than" another, and a similar partial order can be defined for graph grammars. The PLC is formalized in terms of this partial order. These ideas are developed in Chapter 2.

More than just an interesting formalism, the trace grammar representation is the actual intermediate form used by Thinner. Thinner compiles a program into a trace grammar, detects and corrects violations of the principle in that grammar, and then, if necessary, decompiles the reformulated grammar. The trace grammar formalism is used in Chapter 3 to give a formal semantics for the language recognized by Thinner.

How can violations of the principle be detected? This is, of course, an inference problem, one that the reader will not be surprised to learn is undecidable. 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 I developed a new inference technique called *relational constraint.*

Relational constraint starts with a fixed, finite lattice of binary relations on the values that occur during execution of a trace graph. It treats a trace graph as a set of constraints on tuples of binary relations, and it applies these constraints to find the strongest relation it can for any pair of values in the graph. It is, in effect, an $O(n^2)$ forward-chaining inference method. It is combined with more traditional methods of semantic analysis (like the computation of canonical forms for expressions) in Thinner. Relational constraint is discussed in Chapter 4.

How can violations of the principle be repaired? Using the method of inference described above Thinner can find violations of the PLC. These violations are graphs generated in the grammar that Thinner knows how to thin. But knowing how to transform an individual graph isn't enough. The problem is to transform the *grammar* so that it never generates the violation apparent in the example. This is non-trivial because the violation can be generated by a grammar in a variety of ways, perhaps infinitely many.

This *grammar-thinning* problem has an analogue in context-free grammars on strings, and Chapter 5 explores this case. One approximate solution to the string case, the *method of kernels*, also applies to the graph case. This is the transformation method used by Thinner.

My answers to the three questions outlined above are the major contributions of this research, and Thinner is a demonstration that the ideas and methods developed in the following pages do work. Chapter 6 describes the construction of Thinner and presents a selection of timed examples showing some of Thinner's strengths and weaknesses.

The structure of the Thinner project is shown in Figure 1.1.

1.3 Thinner project in context

This section establishes the position of the Thinner project among related research efforts.

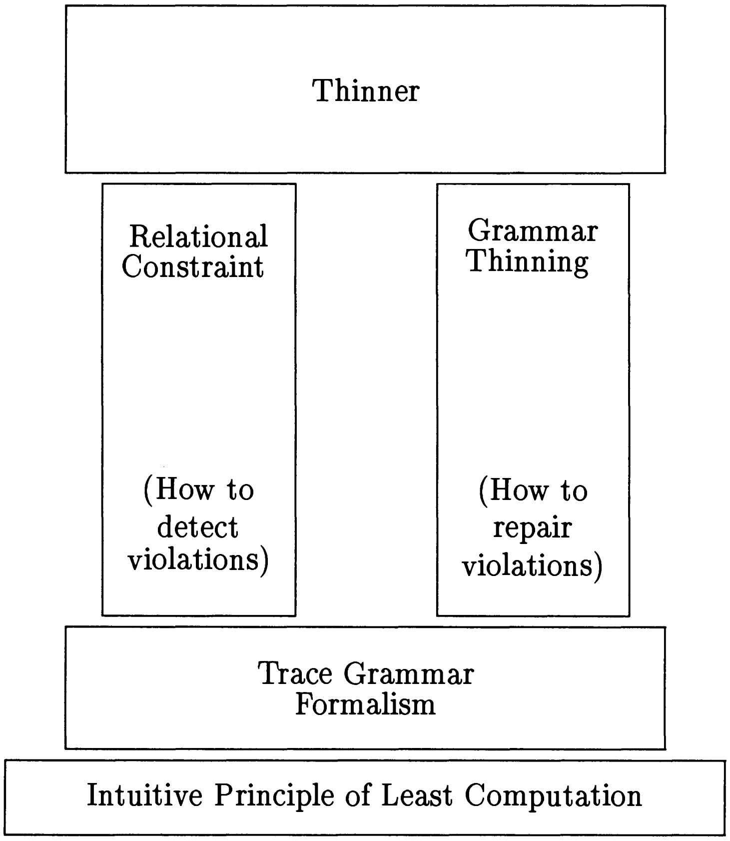


Figure 1.1: The structure of the Thinner project

1.3.1 Irrelevance reformulation

The Thinner project is a descendant of Subramanian's work on irrelevance in first-order theories [Sub89,SG87]. 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 of an irrelevance principle to the problem of eliminating repeated computation in a logic-programming formulation of the Fibonacci function.

1.3.2 Hall's optimizer

The Thinner project is related to the optimizer described in Hall's thesis [Hal90]. I had already completed an early prototype of Thinner when I first heard of Hall's work; and I confess to some alarm when I heard his title, "Program Improvement by Automatic Redistribution of Intermediate Results". It turns out, however, that the two research projects, although undertaken with similar goals, moved in completely different directions.

In discussing his optimizer, Hall appeals to an intuitive principle of optimization similar to my PLC, though he does not derive optimizations from a formal principle as I do. The chief transformation of Hall's optimizer is the redistribution of intermediate values and the removal of disconnected functions in a representation similar to the Plan Calculus [Ric86,RW88], an operation very like the thinning transformation on trace graphs discussed here. 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 correct (except with respect to the given test inputs). The chief transformation, and minor extensions like the addition of new parameters and returned values to subroutines, are introduced with a high-level, intuitive appeal to the underlying principle. In keeping with my goal of showing that optimization can *start* with an optimizing principle, I have taken a more circumscribed approach in the Thinner project. My only input is the raw, purely functional program, and the optimizations produced by the Thinner are guaranteed correct. Within this circumscribed domain I show a clear derivation of optimizations from a rigorous statement of the principle, using a consistent graph-grammar treatment of recursive program structure.

In comparing Hall's work with the Thinner project, it is useful to re-examine the three fundamental questions mentioned above:

- *How can the principle be formalized?* Hall opted to retain an intuitive principle, and used a Plan-Calculus-like intermediate representation. This has the advantage of fitting in well with the existing body of Plan Calculus tools: Hall's optimizer is closer to being a practical tool than Thinner. I chose to develop a new representation and to give a formal expression of the principle being automated. This representation has the advantage of allowing deeper grammatical transformations.

- *How can violations of the principle be detected?* Hall developed very powerful inference techniques, drawing on user-supplied test inputs, optimization invariants, and proofs of correctness. The inference system is not guaranteed correct, but relies on an (unspecified) external verifier. It has the advantage of being able to use information about a program from various sources. The techniques used for inference in Thinner are less powerful, but they have the advantage of being faster, self-contained and correct.

- *How can violations of the principle be repaired?* Hall 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 [Ste90] 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 the method of kernels, which in many cases (including the Fibonacci case) eliminates all instances of unnecessary computation covered by a given example of violation.

1.3.3 Partial evaluation

Expressions whose values are constant certainly violate the PLC, so the Thinner project is related to the considerable body of work on partial evaluation [CD93]. Thinner identifies and eliminates violations of the PLC in programs, so it is, in part, a partial evaluator. But constancy is a special case of PLC violation, and Thinner is a much more general tool.

However, within their own domain, modern partial evaluators are superior to Thinner in several ways. For example, Thinner is not self-applicable. The partial evaluation community has been quite interested in self-application, at least since the 1970's [Ers82] [Ers78] [Fut71] and continuing to the present [JGB+90]. But the language on which Thinner operates (TL, a purely functional, first-order language) is much smaller than the language in which it is implemented (Common Lisp).

Thinner generalizes partial evaluation to a larger class of transformations. In this respect it resembles the "driving" technique used in Turchin's Supercompiler [Tur86,TNT82,Tur80]. This technique arrives at a variety of transformations through symbolic evaluation of expressions in the target language (Refal).[2]

1.3.4 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 [CS70] is the parent of more modern methods [DST80] 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 (remember that we are talking about imperative programs here). Computations may also be partially redundant: redundant on some but not all paths. Morel and Renvoise addressed the problem of eliminating partial redundancies [MR79], and later work built on this idea [RWZ88].

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.

1.3.5 Advanced functional optimization

Other methods of advanced optimization for functional programs include the transformational approach of Burstall and Darlington, the algebraic approach of Backus, and the patterns-of-redundancy approach of Cohen. These are outlined below.

[2] I believe there is a connection between the "kernels" used by Thinner (see Chapter 5) and the configurations taken as "basic" in the Supercompiler. But as the source of basic configurations in Turchin's method is unclear, the connection may be only superficial.

The transformational method was pioneered by Burstall and Darlington [BD77]. 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 these transformations. Field and Harrison comment on this deficiency:

> The choice of the sequence of rules to apply is not precisely specified in the methodology, the next rule to apply in any transformation process being selected by the program designer, perhaps under the guidance of some informal transformation heuristic. [FH88, page 450]

The algebraic approach of Backus is another advanced method of optimization for functional programs [Bac85]. 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 used a similar approach and developed additional theorems in their 1981 paper [KS81]. 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. Is the optimizer based on broad intuitive principles or on an ad hoc collection of transformations? Is it flexible, or does it run through a catalog of known transformations and then quit? These questions depend entirely on the actual transformation theorems and optimization strategies employed; neither of these questions was addressed by Backus, and I believe that they are relatively independent of the algebraic approach.

Another relevant method of optimization is the patterns-of-redundancy approach advanced by Cohen [Coh83]. 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 took a similar approach in their 1987 paper: they were content to identify a specific high-level pattern of redundant computation (Fibonacci-like recursion) [IK87].

1.4 Advice to readers

Readers who want to skim: the first section of each of the following chapters discusses the ideas in that chapter, omitting most of the technical details. The transcripts of Thinner examples in the last section of Chapter 6 are probably also worth skimming.

Readers who are interested in artificial intelligence: Thinner is a step toward an automatic system that can challenge human competence in optimization, and for this reason I do think of it as an AI project. But the techniques that proved most suitable for the task are not standard AI techniques, and you will find few heuristics in the following pages. Do not despair. It would not be difficult to cobble together a collection of optimizing heuristics that performed as well as (or better than) my Thinner on a particular collection of test cases. But what then could you claim for its performance outside that collection? How could you describe the space of transformations of which it was capable? By following a more rigorous path from the formal statement of an optimizing principle, through semantic analysis and grammatical transformation, to design and implementation decisions, we will gain not only a principled optimizer, but also a respectable understanding of its strengths and weaknesses.

Readers who are interested in formal languages: the material in Chapters 2 and 5 should be of most interest, especially the string case of the grammar-thinning problem presented in Chapter 5. This interesting open problem in context-free grammars has potential practical applications.

Readers who are interested in programming language semantics: Chapters 2, 3 and 4 should be of interest. Chapter 4 includes a treatment of the relational constraint inference method in the framework of abstract interpretation.

Readers who are interested in building systems: I recommend the first section of Chapter 2, then Chapters 4 and 6. I believe the relational constraint inference method will be useful in other contexts: it strikes a reasonable bargain between effectiveness and speed, and it is eminently parallelizable.

Readers who want to read everything carefully and tell the author about any errors: bless you.

Chapter 2

Trace Grammars

2.1 Overview

Principled optimization begins with the formal statement of a principle; in this case, the Principle of Least Computation. But first things first: here's an *informal* statement of the principle.

Definition 2.1 (PLC—Informal) *A functional program should not make a computation if:*

1. *the result is already available, being either constant or equal to some other available result; or*
2. *the result is not used, being neither an input to another computation nor an output of the program.*

Naturally, there can be no proof that a formal definition agrees with an intuitive one. There can only be attempts to persuade the intuition. Thinner itself is one kind of persuasion: evidence that, working from the formal definition developed in this chapter, an automatic optimizer can discover many of the optimizations that we normally explain in terms of the informal principle.

Formal statement of the PLC in the domain of purely functional, first-order programs requires a program representation that meets two important requirements. First, it must make all the primitive computations, and all the intermediate values, explicit; a representation that conceals the flow of data from computation to computation will also conceal violations of the PLC. This requirement favors graph-like representations, in which each intermediate value corresponds to an arc, over textual representations. The second requirement is that the representation must not impose any unnecessary structure. Once an optimizer has detected a violation of the PLC, the program transformation that removes that violation should be as straightforward as possible.

To satisfy the requirements of PLC-based optimization I have developed a new representation for purely functional programs [Web92a]. The basis of the 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. I have developed a kind of graph grammar, the *trace grammar*, for this purpose. It is a specialization of the directed, node-label controlled (DNLC) graph grammars studied by Jannsens and Rozenberg [Roz87], structurally (though not semantically) similar to Feder's plex grammars [Fed71] [BH90]. It works this way: a trace graph may contain vertices whose functions are nonterminals, and each nonterminal 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 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 meets the first criterion: it makes each computation and each intermediate value explicit. Thinner compiles purely functional programs directly into trace grammars, which can then be unfolded at will to expose more of the computation to inference and optimization. The trace grammar representation also meets our second criterion, as shown in Figure 2.1. (In this and other trace graph illustrations, computational vertices are represented by rectangles, input and output vertices by lines, and the vertices that fix the values of predicates by squares; dots indicate the input and output arities of each vertex.) The program shown would benefit from code sinking: the evaluation of g is only needed in one branch and should be performed there. The "code movement" normally involved in this optimization is an artifact of the representation: it is required in the Lisp code and in the dataflow graph because they make a commitment to the exact point at which the conditional parts of the computation diverge. The trace grammar representation makes it clear that this optimization merely removes an intermediate computation (the circled vertex) without affecting any other values. In short, it repairs a violation of the PLC, much like dead code removal.

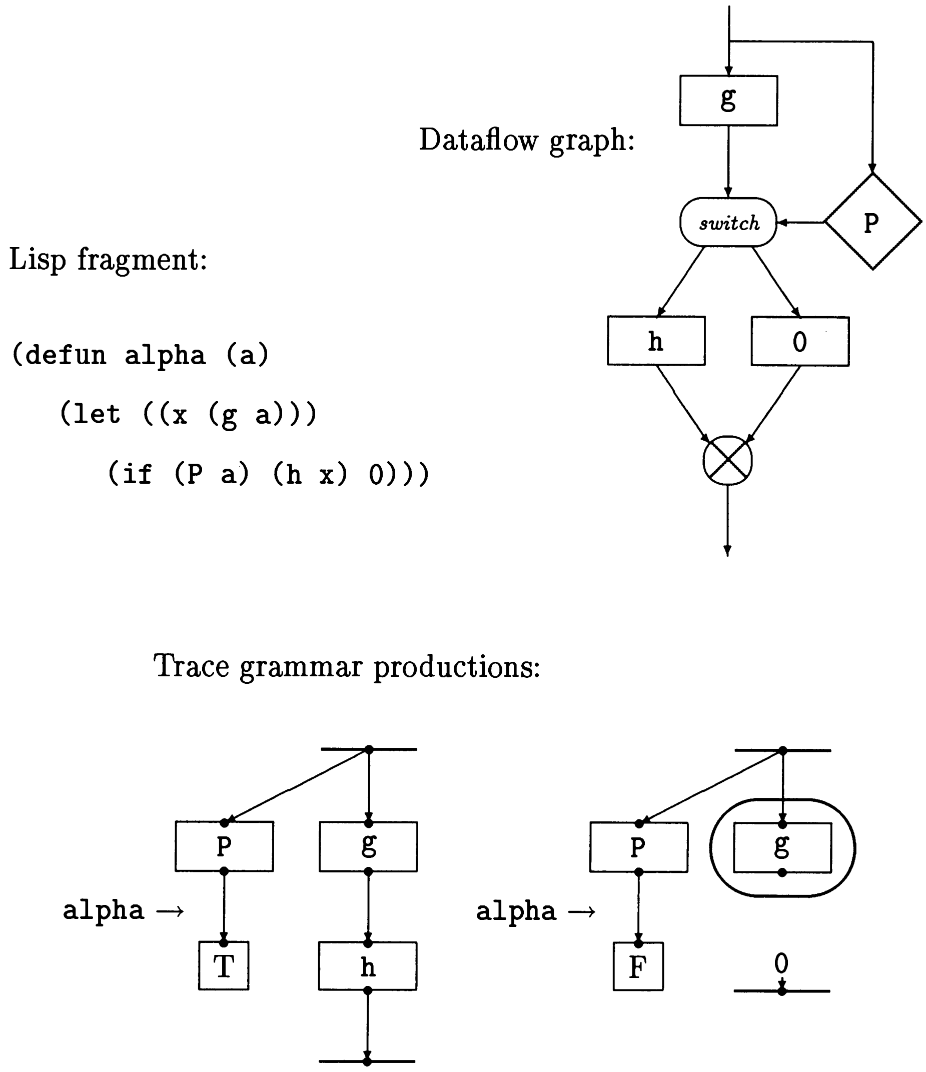


Figure 2.1: A comparison of three representations

The purpose of the trace grammar representation is to allow a rigorous expression of the PLC and to support the automatic derivation of optimizations from it. The PLC sanctions the removal of unnecessary computations from a program; this corresponds to the removal of vertices from a trace graph. Every vertex input in a trace graph must be supplied by an edge, so if you remove vertices you may also have to add edges to bypass the gap. I call this transformation a *thinning.* Obviously, most thinnings alter the graph's function. We are chiefly interested in *conservative thinnings*, which thin a graph without changing the surviving intermediate values.

The conservative thinning transformation on trace graphs induces a relation $\sqsubseteq_c$: $G' \sqsubseteq_c G$ (read "G' is conservatively thinner than G") if and only if G' can be derived from G by removing one or more vertices and adding edges to bypass the gap, so that G and G' agree on every surviving intermediate value. The $\sqsubseteq_c$ relation can be defined similarly for trace grammars, which is the basis of our formal definition of the PLC: for a trace grammar Q there should be no $Q' \sqsubseteq_c Q$. Informally speaking, this principle sanctions the removal of vertices which can be bypassed (which happens when there is some redundant computation, some other source for the same value), as well as the removal of vertices whose outputs are unused and, in the case of trace grammars, the removal of an entire trace graph if it represents a path through the program than can never be taken.

The remainder of this chapter deals with trace graphs and trace grammars, and explores their properties in more detail.

2.2 Trace graphs and the PLC

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. By treating individual paths we avoid, for the moment, a variety of thorny issues about conditional and recursive structures in the source language.

2.2.1 Trace graphs

Definition 2.2 *A* trace graph $G = (V, E)$ *consists of a finite vertex set* V *and an edge set* E*. Each vertex* $v \in V$ *has a fixed* input arity *and a fixed* output arity, *which give the extents of the* input vector v^{in} *and the* output vector v^{out}*. The vertex set* V *is partitioned into five disjoint subsets:*

V_i: *a singleton with input arity 0, standing for the producer of a computation's inputs.*

V_o: *a singleton with output arity 0, standing for the consumer of a computation's outputs.*

V_T: *a set of vertices with input arity 1 and output arity 0. (Each vertex in* V_T *represents a true-or-false decision which, in this path of execution, gets the boolean value true on its input edge. These vertices are called* true predicates.*)*

V_F: *a set of vertices with input arity 1 and output arity 0. (Each vertex in* V_F *represents a true-or-false decision which, in this path of execution, gets the boolean value false on its input edge. These vertices are called* false predicates.*)*

V_C: *a set of vertices with input arity* ≥ 1 *and output arity* ≥ 1. *(These vertices represent primitive functional computations.)*

The edge set E *is an acyclic set of directed edges; each edge is either a pair* (v_i^{out}, w_j^{in}), *representing a value produced by one vertex and required by another, or a pair* (c, w_j^{in}), *representing a constant supplied to a vertex input. There is exactly one edge supplying every vertex input. Each vertex* v *in* $V_C \cup V_T \cup V_F$ *has a vertex label* $\text{label}(v)$.

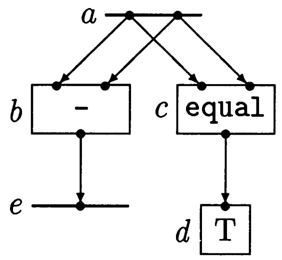
The vertices V_i and V_o represent the external source and sink, the ultimate progenitor of inputs and devourer of outputs. Between V_i and V_o is a computation that maps inputs to outputs by way of intermediate values. A program may contain conditionals, in which case a part of the computation is the evaluation of a boolean function which is used to choose one of two paths for the subsequent flow of data. The trace graph, however, represents a single path of execution: the computation producing the boolean value is present, but the value is consumed by a predicate and the decision, the form of the graph, is fixed.

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 2.3 *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.

Figure 2.2 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.



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

Figure 2.2: Example of a simple trace graph

2.2.2 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: a relation on a space of values which is unspecified but which must include the boolean values true and false. All vertices $v \in V_T$ have label(v) = {(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) = {(false)}. Most vertices in V_C are labelled 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 $x \cdot y \in \text{label}(v)$. Until we introduce an abstract interpretation in Chapter 4, 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. For example, the label of the vertex marked "-" in Figure 2.2 is $\{(x, y, z) \mid x - y = z\}$, and the label of the vertex marked "`equal`" is $\{(x, y, \text{true}) \mid x = y\} \cup \{(x, y, \text{false}) \mid x \neq y\}$.

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

Definition 2.4 *Given a fully terminal trace graph $G = (V, E)$, an* execution of G *is a function f that assigns a value to each vertex input and output such that:*

- *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 2.3 shows an execution of the trace graph from Figure 2.2. (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.

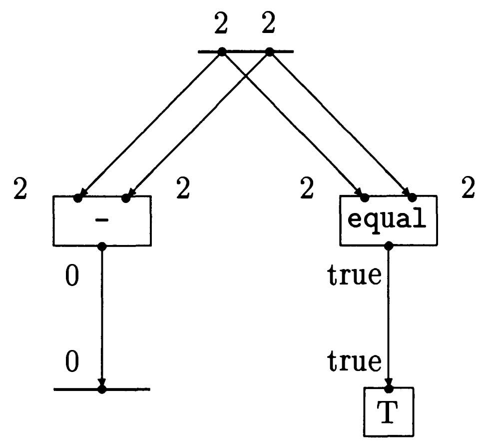


Figure 2.3: Execution of a trace graph on (2,2)

Theorem 2.1 *If all vertex labels are functions, there is at most one execution of a terminal trace graph on any fixed input vector.*

Proof: Consider any execution f of a trace graph G on a fixed input vector x. Take any vertex output v_i^{out}. Since the graph is acyclic, v_i^{out} is at the root of a finite tree of ancestors. There are only two ways a path of ancestry can stop, giving a leaf of the tree of ancestors: it can stop at a vertex input whose edge is from a constant, or it can stop at a vertex output in V_i. (All other vertex inputs are sourced by an edge from a vertex output, and all other vertex outputs are from vertices with input arity ≥ 1.) Now f assigns constant values to all the leaves in the tree: the appropriate constant to vertex inputs with constant sources, and the appropriate value from x to vertex outputs in V_i. Since v_i^{out} is at the root of a tree whose leaves are assigned fixed values and whose internal nodes are functions, f must assign a fixed value to v_i^{out}. Now take any vertex input w_j^{in}. If it has an edge from a constant, f must assign it that constant; if it has an edge from a vertex output v_i^{out}, f must assign it the same value as v_i^{out}, which is fixed for fixed input vector x. Thus, f is determined by x over its entire range. □

2.2.3 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 2.5 *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. I 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.

Definition 2.6 *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 $G' \sqsubseteq^W_F G$.

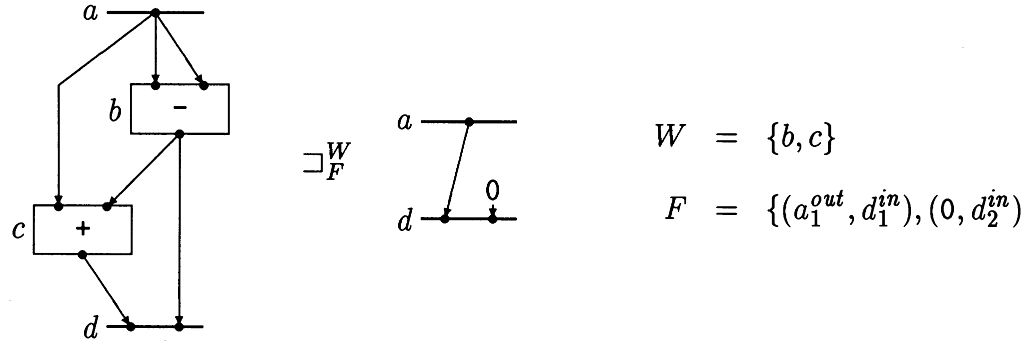


Figure 2.4: A thinning

Figure 2.4 shows an example of a thinning. The following theorem summarizes several properties of the $\sqsubseteq$ relation.

Theorem 2.2 *The following are properties of the $\sqsubseteq$ relation on trace graphs:*

1. *$G \sqsubseteq G$.*
2. *If $G' \sqsubseteq G$ and $G \sqsubseteq G'$ then G and G' are isomorphic.*
3. *If $G'' \sqsubseteq G'$ and $G' \sqsubseteq G$ then $G'' \sqsubseteq G$.*
4. *If $G \sqsubseteq^W_F H$ and $G' \sqsubseteq^{W'}_{F'} H$ then there is a graph G'' such that $G'' \sqsubseteq^{W \cup W'}_{F \cup F'} H$ if and only if no edge in $F \cup F'$ has an end in $W \cup W'$, no two edges in $F \cup F'$ have the same target, and there are no cycles in $(E \setminus E_{W \cup W'} \cup F \cup F')$.*

Proof: The reflexive property follows from the definition, since $G \sqsubseteq^{\emptyset}_{\emptyset} G$. For the identitive property, note that for $G = (V, E)$ and $G' = (V', E')$, $G' \sqsubseteq G$ implies

$|V'| \leq |V|$ and $G \sqsubseteq G'$ implies $|V| \leq |V'|$, so $|V| = |V'|$ and G' is isomorphic to G. To prove the transitive property, suppose $G'' \sqsubseteq_{F''}^{W''} G'$ and $G' \sqsubseteq_{F'}^{W'} G$ for some sets W', W'', F', F''. Construct a set F including every edge in F'', and also every edge in F' with no end in W''. The edge set of G'' contains three groups: those from F'', those from F' not removed by bypassing W'', and those from G not removed by bypassing W'' or W'. By construction the first two groups make up F, and the remaining edges are those not removed by bypassing $W' \cup W''$. So $G'' \sqsubseteq_{F}^{W' \cup W''} G$.

The fourth property gives necessary and sufficient conditions for composing two thinnings by a simple union. We're given that $G \sqsubseteq_{F}^{W} H$ and $G' \sqsubseteq_{F'}^{W'} H$ for some trace graph $H = (V, E)$. Suppose there is a graph G'' such that $G'' \sqsubseteq_{F \cup F'}^{W \cup W'} H$. Since G'' is a trace graph its edge set is acyclic. Also, the edge set meets the two other conditions—there are no two edges to the same target, and no edge with an end in $W \cup W'$—so $(F \cup F')$, which is a subset of the edge set, meets those conditions too.

Now assume all three of the conditions are met for $F \cup F'$; we have to show that $G'' = ((V \setminus (W \cup W')), (E \setminus E_{W \cup W'} \cup F \cup F'))$ is a trace graph. In particular, we have to show that the edge set is acyclic, restricted to the vertex set, and has one edge to every vertex input. It is acyclic by assumption. Also by assumption, no edges involving $W \cup W'$ were added; and all old edges involving $W \cup W'$ were removed. The new edge set has at least one edge to every vertex since every vertex that had an input edge from W or W' is the target of a bypass edge in $F \cup F'$; and it has no more than one edge to any vertex since there are no two edges in $F \cup F'$ with the same target but different sources. So G'' is a trace graph. □

Usually, when you thin a graph, you end up with a graph that computes some completely different partial function. That isn't going to be useful in an optimizer!

We're really interested in a special class of thinnings: those that preserve functionality in a particular way. The two graphs of Figure 2.4 already stand in this (much more demanding) relation. Executions of the two graphs on the same value always agree.

Definition 2.7 *Suppose $G = (V, E)$ and $G' = (V', E')$ are terminal trace graphs. $G' \sqsubseteq_c G$ 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, $G' \sqsubset_c G$.

A conservative thinning removes vertices from a graph in a way that preserves the values assigned to the remaining vertices by all executions.[1] 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 that by Definition 2.7, 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. Theorem 2.3 lists a variety of other properties of $\sqsubseteq_c$.

Theorem 2.3 *The following are properties of the $\sqsubseteq_c$ relation on trace graphs:*

1. *$G' \sqsubseteq_c G$ implies $G' \sqsubseteq G$ (but not the reverse).*

[1] Johnson [Joh86] 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 my "conservative thinning" only accidentally; see the proof of Theorem 2.5, below.

2. $G \sqsubseteq_c G$.

3. If $G' \sqsubseteq_c G$ *and* $G \sqsubseteq_c G'$ *then* G *and* G' *are isomorphic.*

4. If $G'' \sqsubseteq_c G'$ *and* $G' \sqsubseteq_c G$ *then* $G'' \sqsubseteq_c G$.

Proof: The first three properties follow from the definition and from the corresponding properties of $\sqsubseteq$. The transitive property follows from the transitive property of $\sqsubseteq$: when $G'' \sqsubseteq_c G'$ and $G' \sqsubseteq_c G$, we know that $G'' \sqsubseteq G$, and the two subgraph isomorphisms can be composed to give a mapping function h that satisfies the definition for $G'' \sqsubseteq_c G$. □

2.2.4 The PLC for trace graphs

The PLC for the domain of trace graphs has a natural formulation in terms of conservative thinning.

Definition 2.8 (PLC—Trace Graphs) *For a trace graph* G *there should be no* $G' \sqsubseteq_c G$.

Compare this definition with the informal statement of the PLC in Definition 2.1. The first clause of the informal statement says that a computation is unnecessary if the result is constant or equal to some other available result. Replacement by a constant or by a different available result is the idea captured by the bypass set—we've simply defined "availability" in terms of legal trace graph structures so that, roughly speaking, a result is available as long as using it doesn't introduce a cycle. The second clause of the informal statement says that a computation is unnecessary if the result is unused. This also is captured by the formal definition, since if no edges leave the set W, it is vacuously true that a bypass exists ($F = \emptyset$) and that the resulting thinning is conservative.

2.2.5 Discussion

Any optimization principle like the PLC must refer to a cost model. The PLC's cost model is a simple one: the cost of a trace graph is the number of vertices. This model would be inadequate for parallel computation, because one graph can be thinner than another yet have a longer critical path.

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 Perlin [Per89]. 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 2.4 *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: Using the vertex types mentioned above we can construct a graph component for any expression E that involves only integer addition, multiplication, exponentiation, positive constants, and variables (the variables are input edges at the leaves of the expression tree). So given any two expressions E_1 and E_2 we can construct a component that evaluates both and compares the two results for integer equality. Let G be a graph with such a structure for expressions E_1 and E_2, in which V_i is connected to the expression inputs, and the comparison output is connected to a false predicate $v \in V_F$. This G has an execution on x if and only if x is an assignment to the variables of the E_1 and E_2 that makes them unequal. Vertex v is unnecessary if and only if $E_1 = E_2$ is unsatisfiable. But this is undecidable: there is 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 [JM84]. □

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 2.9 *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 2.5 *The minimum thinning problem is NP-complete.*

little further significance. The practical 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 practical reformulation problem involves deep difficulties in transforming recursive structures, 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.

2.3 Sets of trace graphs and the PLC

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.

2.3.1 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 2.5. It has exactly one

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 [GJ79].

An instance of the minimum cover problem is a collection C of subsets of a finite set S, and a positive integer $k \leq |C|$. We'll 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 we 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 two problems—identifying violations of the PLC and repairing them—are treated in Chapters 4 and 5. But the uncomputability of the first and the intractability of the second turn out to have 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.

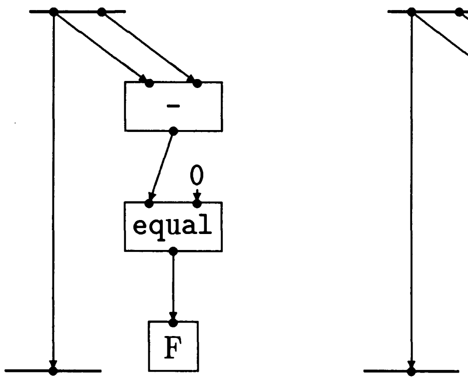


Figure 2.5: A non-executable set of trace graphs

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*.

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 2.10 *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 2.11 *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 2.12 *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. Intuitively, any program that makes all its decisions based on predicates 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 2.6 *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 by Theorem 2.1 $f'(v_1'^{in}) = f(v_1^{in})$. This is a contradiction. □

Theorem 2.6 shows that any executable set is deterministic. As Figure 2.5 shows, the reverse is not true.

2.3.2 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.[2] That is:

Definition 2.13 *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.*

Theorem 2.7 summarizes some properties of the $\sqsubseteq$ relation on sets.

Theorem 2.7 *The following are properties of $\sqsubseteq$ for sets of trace graphs:*

1. $S \sqsubseteq S$.
2. *If* $S'' \sqsubseteq S'$ *and* $S' \sqsubseteq S$ *then* $S'' \sqsubseteq S$.
3. $S' \subseteq S$ *implies* $S' \sqsubseteq S$.

Proof: For transitivity, $S'' \sqsubseteq S'$ means that for every $G'' \in S''$ there is some $G' \in S'$ with $G'' \sqsubseteq G'$, and $S' \sqsubseteq S$ means that for every $G' \in S'$ there is some $G \in S$ with $G' \sqsubseteq G$. It follows that for every $G'' \in S''$ there is some $G \in S$ with $G'' \sqsubseteq G$, so $S'' \sqsubseteq S$. The third property follows directly from the definition: if $S' \subseteq S$ then for every $G' \in S'$ there is some $G \in S$, namely $G = G'$, with $G' \sqsubseteq G$. This also implies reflexivity since $S \subseteq S$. $\Box$

Note that we have lost the identitive property of the $\sqsubseteq$ relation on individual trace graphs: two sets can be strictly thinner than each other.

The $\boxed{\sqsubseteq}$ 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

[2] Compare this with the lower or Hoare powerdomain [GS90].

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 2.14 *Suppose S and S' are sets of trace graphs. $S' \sqsubseteq_c S$ 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, $S' \sqsubset_c S$.

Note that this definition gives $\{G'\} \sqsubseteq_c \{G\}$ if and only if $G' \sqsubseteq_c G$, 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 not be represented in the thinner set. The fact that the $\sqsubseteq_c$ 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" style transformations.

Theorem 2.8 summarizes some other properties of the $\sqsubseteq_c$ relation for sets.

Theorem 2.8 *The following are properties of $\sqsubseteq_c$ for sets of trace graphs:*

1. *$S' \sqsubseteq_c S$ implies $S' \sqsubseteq S$ (but not the reverse).*
2. *$S \sqsubseteq_c S$.*
3. *If $S'' \sqsubseteq_c S'$ and $S' \sqsubseteq_c S$ then $S'' \sqsubseteq_c S$.*

Proof: The first property follows directly from the definition. For every $G \in S$ there is some $H \in S$, namely $H = G$, for which $H \sqsubseteq_c G$. This proves the reflexive property. Transitivity follows because for each $G'' \in S''$ and each execution f'' of G'', there is a thicker $G' \in S'$ with an execution f' that agrees with f'', and there is a thicker $G \in S$ with an execution f that agrees with f'. The two subgraph isomorphisms can be composed to give a mapping function h from G'' to G that satisfies the definition, and the transitivity of $\sqsubseteq$ completes the proof. □

If $S' \sqsubseteq_c S$ and 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. How can we characterize $\sqsubseteq$ relations that preserve executability?

It is not hard to characterize an executable thinning if it preserves a decision tree T of the original graph. It removes no graph from the set and no predicate from any graph; and if it changes an ancestor of a predicate in any graph, it must make the same change to every graph at that node of T where the predicate is used (since all the predicates in a predicate partition must match). A similar, more complicated characterization can be developed to cover executable thinnings that only prune a decision tree of the original graph.

But an executable thinning need not have a decision tree that bears any relation to a decision tree of the original graph. Consider the set thinning shown in

Figure 2.6. By forming differences among formerly identical predicates it excludes the original decision tree, and by erasing differences among predicates it allows a new decision tree. So although every graph in the thinned set is a thinned version of a graph in the original, the decision trees for the thinned set may be completely different—they need not be "thinned" versions of the original decision trees.

2.3.3 The PLC for executable sets

The principle of least computation for individual trace graphs expressed in Definition 2.8 extends naturally to executable sets of trace graphs.

Definition 2.15 (PLC—Executable Sets) *For an executable set S there should be no executable $S' \sqsubseteq_c S$.*

We started with an intuitive idea: that a program should not do anything unnecessary. Elaborated on an intuitive level, this meant that a program should not make a computation with a constant, redundant or unused result. Formalized in terms of individual trace graphs, it meant that there should be no way to remove vertices, bypass them, and end up with a legal trace graph that makes the same computation for any input but with fewer intermediate values. Now we have reached the level of executable sets, and at this level the principle is that there should be no way to remove and bypass vertices and end up with a set that makes the same computation for any input, with fewer intermediate values for at least one.

Actually, that isn't quite enough to make a violation of the PLC as expressed in Definition 2.15. The principle is only violated when there is a conservatively thinner *executable* set. .There may be superfluous computation that cannot be removed without making the set non-executable, as is the case in Figure 2.7. There are several non-isomorphic sets conservatively thinner than the set $\{G, H\}$ shown—

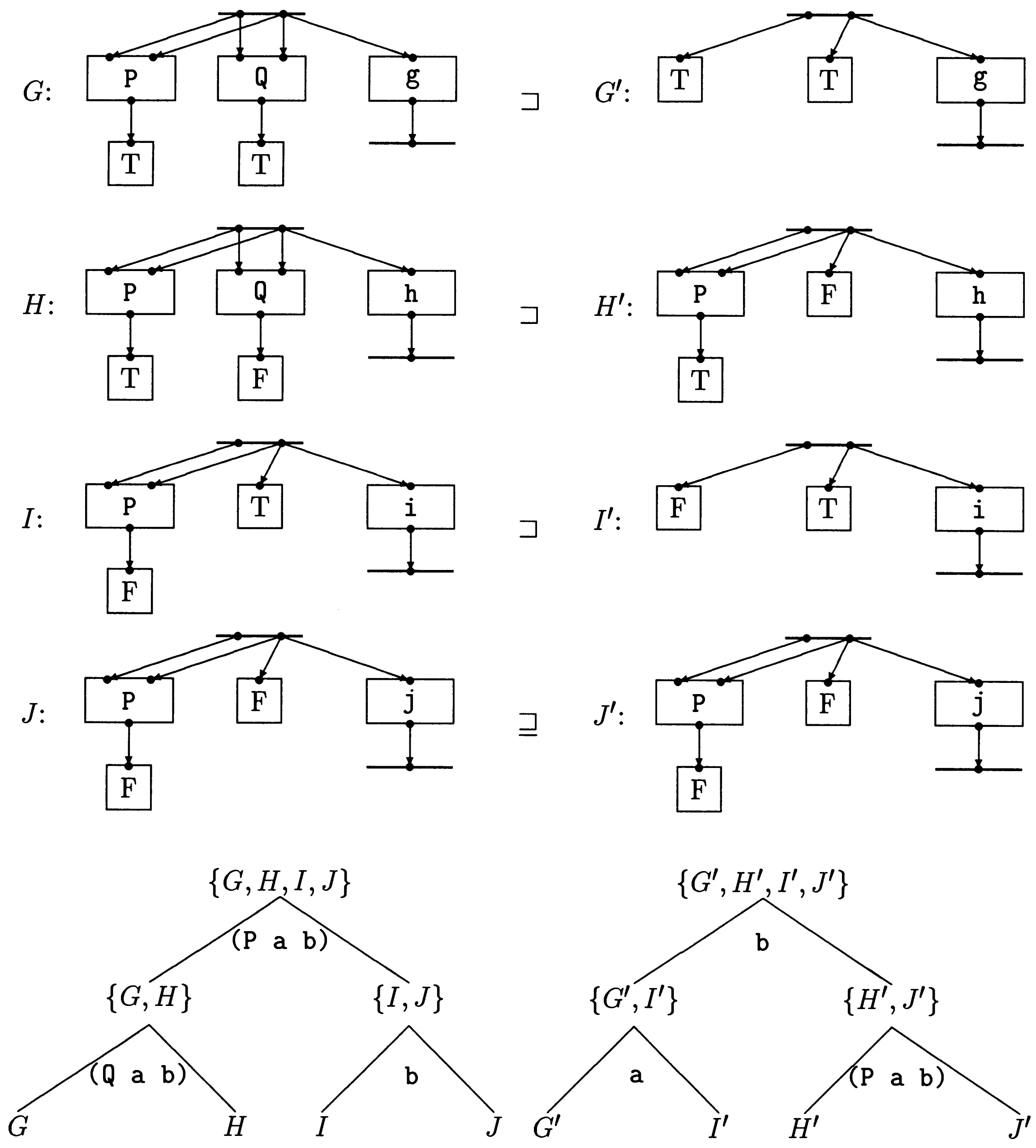


Figure 2.6: Trace graph sets $\{G, H, I, J\} \sqsupset \{G', H', I', J'\}$, with decision trees

one of them is $\{G', H\}$, with G' as indicated—but none is executable. In effect, there is no way to discover that graph G is the one with an execution on a given input until after performing computations that are only necessary in H.

2.3.4 Discussion

There is an interesting connection between sets of trace graphs that are not necessarily executable and nondeterministic automata. Perhaps this would be a fruitful area for further research.

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. In the next section we will see that recursive structure makes these simple concepts much harder to manipulate.

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 doesn't 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 [AN87], in which the two arms of a conditional have inputs directed to them by "switch" nodes and outputs collected from them by "mux" nodes. In the introduction of this chapter, Figure 2.1 showed an illustrative code fragment expressed both as a dataflow graph and as an executable set.

The code fragment of Figure 2.1 exposes an opportunity for the optimization known as code sinking: the evaluation of g is only needed on the left branch and should be performed there. The "code movement" required by this transformation in the dataflow case is an artifact of the representation; it is necessary because the representation has made a commitment to the exact point at which conditional

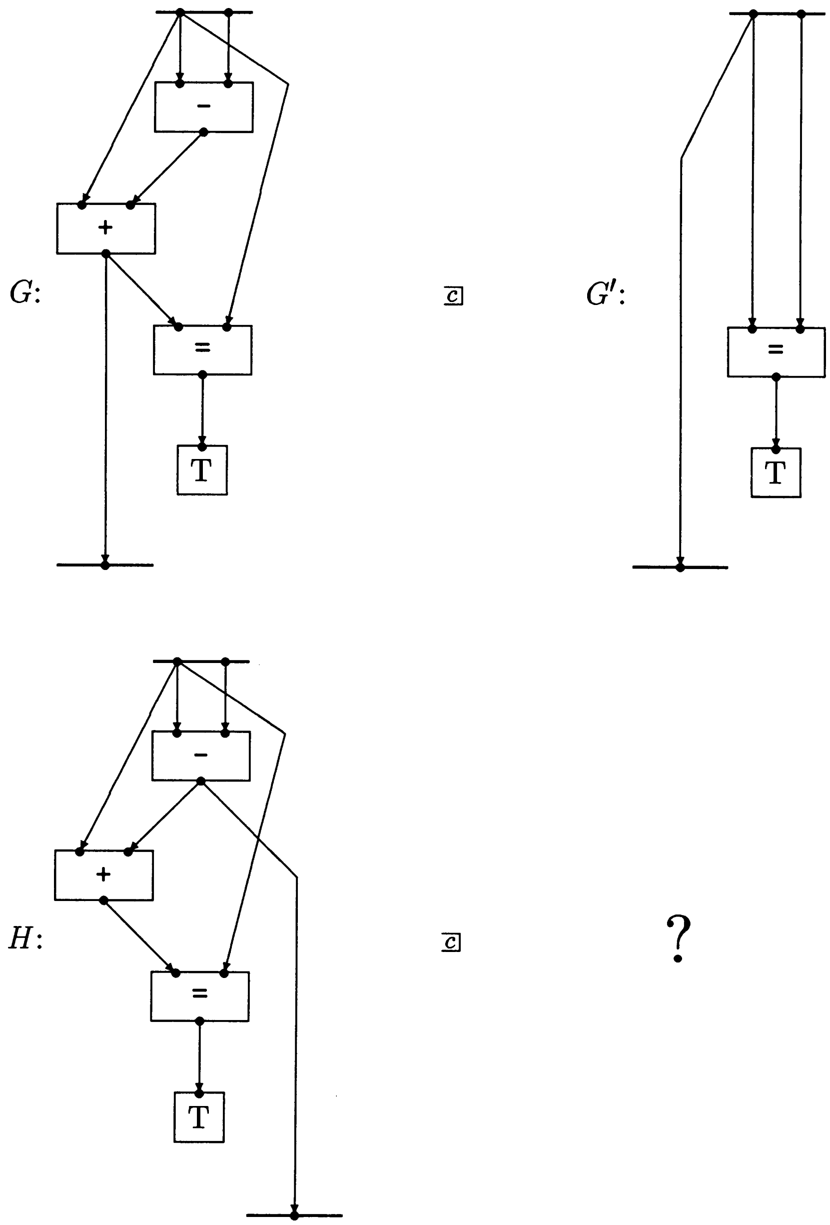


Figure 2.7: A set $\{G, H\}$ not quite in violation of the PLC

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, much like dead code removal. Figure 2.8 shows another such comparison; again, the dataflow graph representation obscures an obvious thinning.[3]

2.4 Trace grammars and the PLC

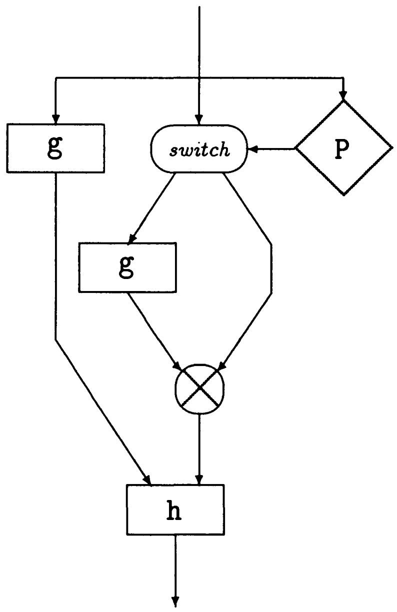
Most interesting programs have infinitely many possible paths of execution, so the explicit trace graph set is going to be inadequate as a representation. What we need is a finite representation that implicitly identifies a potentially infinite trace graph set. We therefore turn to trace grammars, which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

2.4.1 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

[3]These examples are intended to explain the choice of executable sets as a formalism for study of the PLC, not to criticize the ideas of dynamic dataflow. Dynamic dataflow was developed for a different purpose, namely, as an architectural approach for exposing fine-grained parallelism.

Dataflow graph:



Lisp fragment:

```
(defun alpha (a)
   (h (g x) (if (P x) (g x) x)))
```

Executable set:

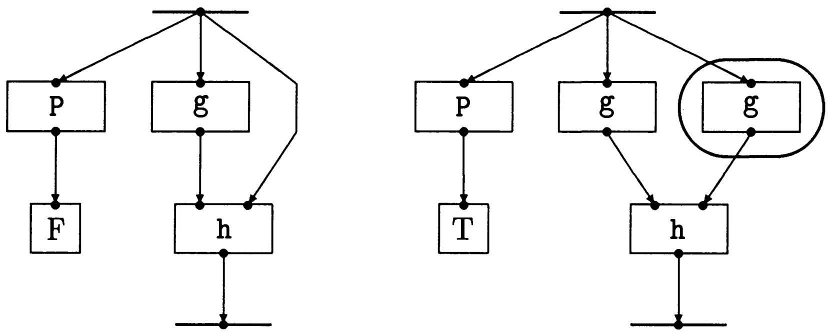


Figure 2.8: Another comparison of three representations

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:

Definition 2.16 *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 $E' = E_{\text{old}} \cup E_{\text{seam}} \cup E_{\text{new}}$.*

Vertex replacement is a straightforward operation because the graph inputs and outputs match up with the vertex inputs and outputs. There's no uncertainty about how a graph should be substituted for a vertex: just connect the matching inputs and outputs. Figure 2.9 shows an example of a vertex replacement.

Definition 2.17 *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 $(\alpha \to D)$, 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 $N \cup T$.*

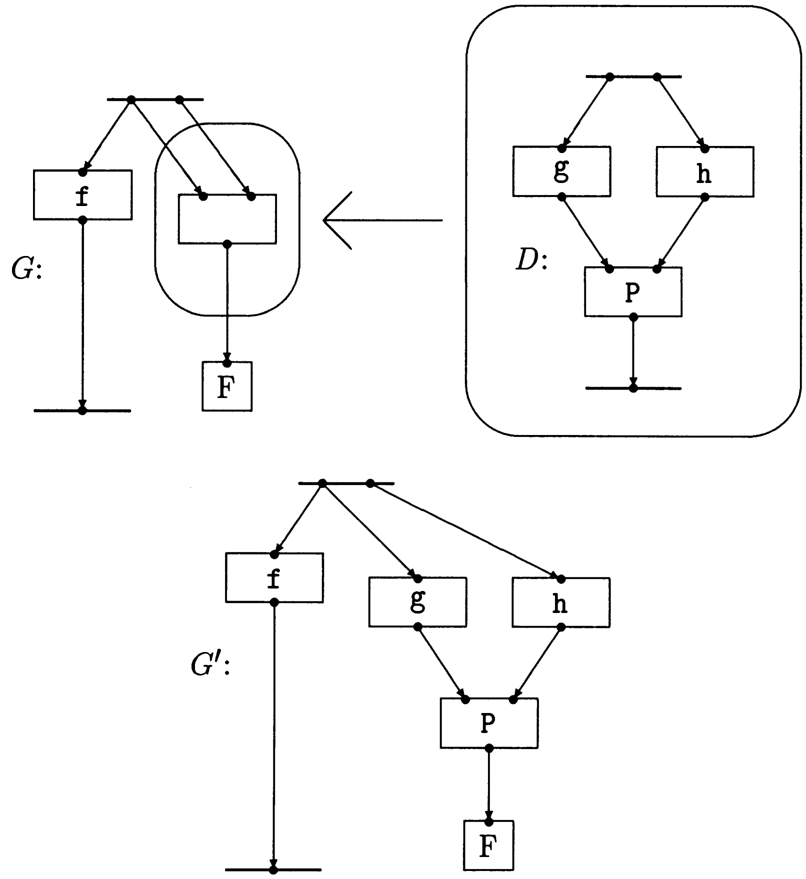


Figure 2.9: The replacement of a vertex in G by a graph D

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 to define the set of trace graphs generated by a trace grammar:

Definition 2.18 *Let $Q = (N, T, P, S)$ be a trace grammar, and let A be the set of graphs G such that $S \Rightarrow_Q^* G$ and all vertices in G have functions in T. A* set generated by *Q, denoted $L(Q)$, is a minimal set of vertex-disjoint trace graphs with an element isomorphic to every element of A.*

Figure 2.10 shows an example of a trace grammar with part of the set generated by it. This grammar generates an executable set, which not all grammars do. There is a restriction on the set of productions that guarantees that the generated set will be executable. Trace grammars that obey this restriction, like the one in Figure 2.10, are *normal* trace grammars.

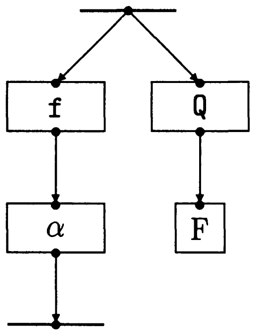
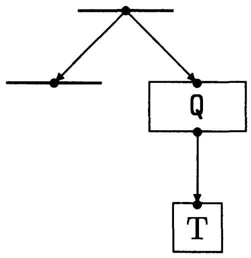
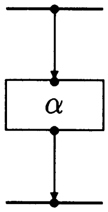
Definition 2.19 *A* normal *trace grammar $Q = (N, T, P, S)$ is one in which all nonterminals α in N have the property that $\{D \mid (\alpha \rightarrow D) \in P\}$ is an executable set.*

Theorem 2.9 *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

A trace grammar $Q = (N, T, P, S)$:

$N = \{\alpha\}$ $T = \{\texttt{f}, \texttt{Q}\}$

$P = \{(\alpha \rightarrow$  $), (\alpha \rightarrow$  $)\}$ $S =$ 

The first three elements of the generated set:

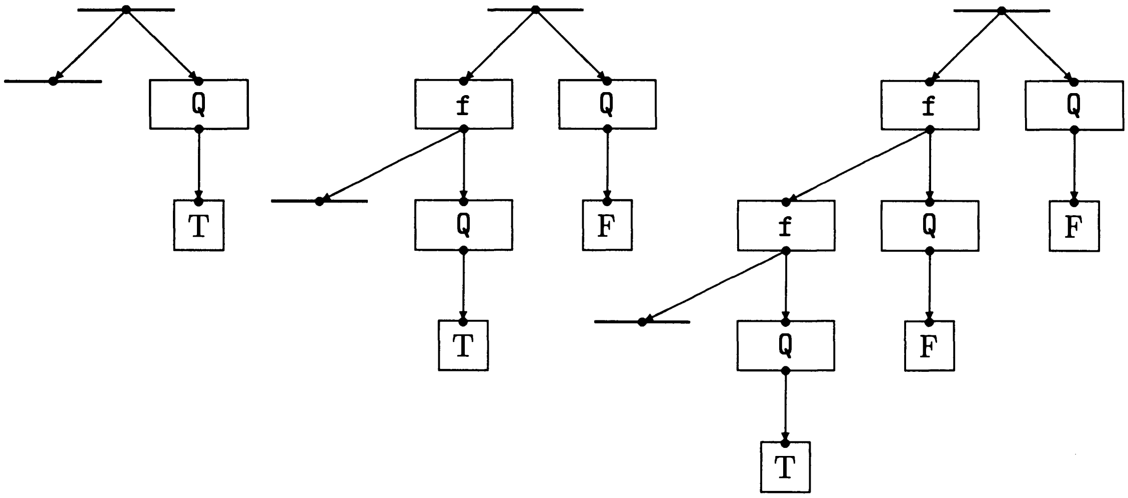


Figure 2.10: Example of a trace grammar and generated set

of G.[4] 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. $\Box$

2.4.2 Correspondence to CFG's

Can any executable set be generated by some trace grammar? To answer this question, we will develop a connection between trace grammars and traditional context-free grammars on string alphabets. This connection will also prove useful in answering other questions about trace grammars later on.

First we'll define a correspondence between symbols in an alphabet Σ and trace graph components. Let Σ^+ be $\Sigma \cup \{\omega\}$, where ω is a new symbol not occurring in Σ. (ω will mark the end of a string.) Let P_i denote a distinct function for each i from 1 to $\lceil \log_2 |\Sigma^+| \rceil + 1$. For each symbol x in Σ^+, define a unique truth assignment vector v^x so that v^x_i is x's assignment to P_i and no other symbol has

[4] 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.

the same pattern of truth assignments. Finally, let f^x denote a distinct function for each symbol x in Σ^+. The graph component corresponding to a symbol x in Σ^+ is as pictured in Figure 2.11. It contains a vertex for each of the P_i's, each with an edge to a vertex either in V_T or V_F as determined by the unique truth assignment for x. It also has a vertex for the function f^x. The inputs to the component are the inputs to f^x and the P_i's. They have a common source as indicated in the diagram. The output of the component is the output from f^x.

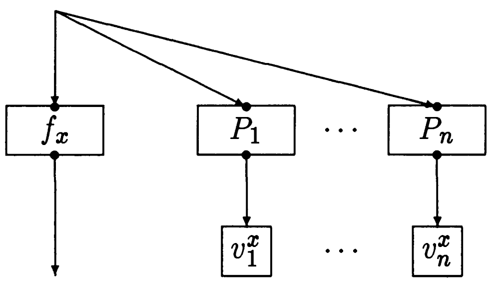


Figure 2.11: The trace graph component corresponding to $x \in \Sigma^+$

This correspondence between symbols and graph components leads to a simple correspondence between strings and graphs. If s is a string of length n, index the characters of the string as $s_1 \ldots s_{n+1}$, where s_{n+1} refers to the string end marker ω. A graph $G = (V, E)$ corresponds to the string s if and only if:

- G contains a unique component for each s_i, an input vertex, an output vertex, other edges as specified below, and nothing else.
- There is a vertex in V_i with a single output connected to the input of the component corresponding to s_1.
- For every i in the range $1 \leq i \leq n$, the output of the component corresponding to s_i is connected to the input of the component corresponding to s_{i+1}.

- The output of the component corresponding to s_{n+1} is connected to the single vertex input of the vertex in V_o.

It should be clear there is a one-to-one correspondence between strings on the alphabet Σ and graphs as described above. Sets of such graphs correspond to languages over Σ.

Theorem 2.10 *If S is a set of graphs that correspond to strings, and if S has no two elements isomorphic to each other, then S is executable.*

Proof: We proceed by induction on the length n of the longest string corresponding to a graph in S. In the base case $n = 0$: since two graphs corresponding to the empty string would be isomorphic to each other, S must be a singleton and hence has a decision tree. For $n > 0$, note that every graph G in S has a matching collection of P_i's connected directly to V_i. So there is a partial decision tree that partitions on each P_i in turn (any order will do). Since each x has a unique truth assignment to the P_i's, the partitions at the bottom of this partial tree are sets S^x whose graphs agree that the first component is an x for each $x \in \Sigma^+$. There is a decision tree for S^ω since, as noted before, it can contain at most one graph. There is also a decision tree for each S^x with $x \in \Sigma$, since we can omit the identical first component from each graph, get a decision tree for that set (which must exist by the inductive hypothesis), and use that tree for S^x. Since there is a partial decision tree for S and a decision tree for each leaf of that partial tree, there is a full decision tree for S. $\square$

Theorem 2.11 *Suppose a string language M corresponds to a set of trace graphs R. M is context free if and only if R is generated by a trace grammar.*

Proof: The "only if" direction is the easier of the two: given a CFG Q with nonterminal set N, terminal set T, productions P and start symbol S, construct a trace grammar $Q' = (N', T', P', S')$ as follows. Extend the correspondence between strings and graphs to include strings of terminals and nonterminals: a nonterminal in a string corresponds to a nonterminal function in the graph. N' contains a nonterminal function f^V for every $V \in N$. T' contains a terminal function f^x for every $x \in T$. Now for every production $(V \rightarrow s_1 \dots s_n) \in P$, P' contains a production $(f^V \rightarrow G)$ where G is the graph corresponding to the string $s_1 \dots s_n$. Finally, start graph S' is simply the graph corresponding to the string S (so it has a nonterminal function f^S followed by the component for ω). By construction, if G' is a graph corresponding to a string g, $G' \Rightarrow_{Q'} H'$ if and only if H' corresponds to a string h for which $g \Rightarrow_Q h$. Since the start graph S' corresponds to the string S, it follows that $S' \Rightarrow^*_{Q'} G'$ if and only if $S \Rightarrow^*_Q g$, where g is the string corresponding to G'. Hence, the set generated by Q' corresponds to the language generated by Q.

The other direction is more difficult, since there is no guarantee that a given trace grammar is in a nice linear form. All we know is that every graph in the generated set corresponds to a string; the intermediate forms in the derivation need not. We will assume that the given trace grammar has no useless nonterminals (that is, that all nonterminals can be expanded into terminal graphs). This is without loss of generality since a trace grammar with useless nonterminals can easily be converted into one without them, by essentially the same method that removes useless nonterminals from context-free grammars [HU79].

So suppose $Q' = (N', T', P', S')$ is a trace grammar that generates a set of trace graphs corresponding to strings, and suppose N' contains no useless nonterminals. Define a CFG $Q = (N, T, P, S)$ as follows. N contains a nonterminal symbol α^i_j

for each nonterminal function $\alpha \in N'$ and each input i to α and each output j from α. T contains a terminal symbol x for each $f^x \in T'$. Now a path in a trace graph corresponds to a string over $T \cup N$ as follows: each terminal function f^x corresponds to an x in the string, each nonterminal function α entered in the path at input i and exited at output j corresponds to an α^i_j in the string. For each production $(\alpha \to D) \in P'$, and for each input i to α and output j from α, and for each path q in D from input i to output j, P contains a production mapping α^i_j to the string corresponding to the path q. P also contains productions for the start symbol S: one for each path from the input to the output in the start graph S'. (If S' does not have a single input and a single output then no generated set could correspond to a string, so the generated set must be empty and corresponding language context free. We can therefore assume that S' has a single input and a single output.)

Now $S \Rightarrow^+_Q g$ if and only if $S' \Rightarrow^*_{Q'} G'$, where G' contains a path from input to output corresponding to g. There is a simple proof of this by induction on the length of the derivation, which we will skip.

It now remains to be shown that the language of Q corresponds to the set generated by Q'. Suppose a string g is in the language of Q. Then, as shown above, Q' derives a graph G' with a path of terminal functions from input to output corresponding to g. We cannot assume that G' is fully terminal, but since Q' contains no useless symbols, a terminal graph H' can be derived from G'. This H' is in the set generated by Q', so by assumption it must correspond to a string; and since it has the same path G had from input to output, it must in fact correspond to s. In the other direction, suppose G' is one of the (fully terminal) graphs generated by Q'. Since G' corresponds to a string, it has exactly one path from input to output; and as shown above, Q derives the terminal string corresponding to that path. □

Now we can answer the question of whether every executable set is generated by some trace grammar. Every string language L corresponds to some set of trace grammars L' in which no two elements are isomorphic to each other. By Theorem 2.10, L' is executable. But by Theorem 2.11, L' has a trace grammar only if L is context free. So not every executable set has a trace grammar: if L is not a context-free language, the corresponding set L' is executable but has no trace grammar.

The relation between trace grammars and CFG's also gives a reduction which can be used to show that several questions about trace grammars are undecidable. Among these:

- Are the sets generated by two trace grammars disjoint?
- Do two trace grammars generate isomorphic sets?
- 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 nonterminal 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.

2.4.3 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 2.20 *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 2.11 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 $\text{dom}(S) = \emptyset$, 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. One potentially useful characterization would be as a class of transformations to trace grammars: some way to transform a grammar into any thinner grammar. Some idea of the kind of transformational power required for such a characterization can be gained by examining the consequences of Theorems 2.7 and 2.11. For any alphabet Σ the language Σ^* is context free, and so the corresponding executable set has a linear trace grammar Q—quite a simple one at that. Now any context-free language M over Σ is a subset of Σ^*; and so the executable set corresponding to M is a subset of $L(Q)$. But any subset of a set of trace graphs is a thinning! This means that the class of thinning "transformations" of trace grammars has at least enough power to generate any grammar for a set of graphs that correspond to strings from a grammar for the set that corresponds to Σ^*—which is to say, at least enough power to generate any context-free grammar out of practically nothing.

Is the $\sqsubseteq$ relation on trace grammars decidable? I do not know. However, there is an interesting corresponding question about CFG's. Define $L(C)$ to be the language of a CFG C, and define 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 construction of Theorem 2.11 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.[5] So perhaps the $\sqsubseteq$ relation is decidable too.

2.4.4 The PLC for trace grammars

We can express the PLC for executable trace grammars in terms of the sets they generate.

Definition 2.21 (PLC—Trace Grammars) *For an executable trace grammar Q there should be no executable $Q' \boxed{\sqsubset} Q$.*

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.

2.5 Applying the formal principle

Now we have a formal statement of the PLC. How can can it be embodied in an automatic optimizer? We can't just search for conservatively thinner grammars! One observation is that if a grammar violates the PLC for trace grammars, it

[5]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 [Cho62].

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 (not necessarily terminal) graphs generated in the grammar, looking for violations of the PLC. (Chapter 4 deals with the problem of analyzing individual trace graphs.) 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. (Chapter 5 treats this aspect of the problem.) This step 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's 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 $\sqsubseteq$ 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 $\sqsubseteq$ relative to a mapping from nonterminals to partial functions. If H and G are trace graphs containing nonterminals and $\mathcal{F}$ is a function mapping each nonterminal to a function, and if that mapping gives $H \sqsubseteq G$, we'll say that $H \sqsubseteq G$ *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 $\mathcal{F}$: 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: remove one in a thinning and 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 is the approach Thinner takes—it just doesn't make sense to waste any computational effort on ensuring that the set of inputs on which the program diverges doesn't shrink.

2.5.1 Weak conservative thinning

To talk about this more formally, let's define *weak conservative thinning*: thinning that is almost conservative, except that it may enlarge the domain. In Definitions 2.7 and 2.14 we insisted that conservatively thinner graphs, sets and grammars have the same domains as their thicker counterparts. By omitting the third condition from these definitions, we can define a relation $\sqsubseteq^{c}_{\mathrm{weak}}$ which allows thinner graphs, sets and grammars to have larger domains.

Definition 2.22 *Suppose $G = (V, E)$ and $G' = (V', E')$ are terminal trace graphs. $G' \sqsubseteq^{c}_{\mathrm{weak}} G$ 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 $f' = f \circ h$.*

Suppose S and S' are sets of trace graphs. $S' \mathrel{\sqsubseteq\!\!\!\!\!\!{\scriptstyle c}\;}_{\mathbf{weak}} S$ 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 $f' = f \circ h$.*

2.5.2 Strong conservative thinning of right-hand sides

Let's look at strong conservative thinning first. We want to be able to gather the right-hand sides of a nonterminal into a set (which will be executable, since the grammar is normal), thin that set conservatively, and conclude that we have a conservatively thinner grammar. This seems reasonable, and it's correct; but it's not simple to prove.

Here we go. In the following discussion, let $Q = (N, T, P, S)$ be a normal trace grammar. For some nonterminal $\alpha \in N$ let A be the set $\{D \mid (\alpha \rightarrow D) \in P\}$, and suppose there is some executable set A' such that $A' \mathrel{\sqsubseteq\!\!\!\!\!\!{\scriptstyle c}\;} A$ relative to every mapping $\mathcal{F}$ from nonterminals to partial functions. Let Q' be a trace grammar formed from Q by removing the old productions for α and adding a new production $(\alpha \rightarrow D')$ for every $D' \in A'$. Because Q is normal each nonterminal corresponds to a partial function, obtained as the executable set generated from that nonterminal under Q. Define $\mathcal{F}_1$ to be this mapping from nonterminals to their actual functions under Q.

Lemma 2.12 *Suppose we have a graph G' and set R, possibly containing nonterminals, for which $\{G'\} \mathrel{\sqsubseteq\!\!\!\!\!\!{\scriptstyle c}\;} R$ relative to $\mathcal{F}_1$, and suppose G' derives some terminal graph I' in Q'. Then R derives some terminal set T in Q for which $\{I'\} \mathrel{\sqsubseteq\!\!\!\!\!\!{\scriptstyle c}\;} T$.*

Proof: by induction on the number n of steps in the derivation of I'. In the base case $n = 0$, so G' is terminal and $I' = G'$. R need not be terminal since its graphs are thicker than G, but let S be the set of terminal graphs generated from R under Q. Since $\{I'\} \sqsubseteq R$ relative to $\mathcal{F}_1$, $\{I'\} \sqsubseteq S$ relative to $\mathcal{F}_1$; and since all the graphs are fully terminal, $\{I'\} \sqsubseteq S$.

In the inductive case $n > 0$: let H' be the result of the first step in the derivation of I', so that $G' \Rightarrow_{Q'} H' \Rightarrow^*_{Q'} I'$. If the first step is the expansion of some nonterminal $\beta \neq \alpha$, let S be the set that results from applying that same step to every member of R. (All members of R are thicker than G, so they all have a corresponding β to expand.) Since $\{G'\} \sqsubseteq R$ relative to $\mathcal{F}_1$, and since β has been replaced on both sides with a subgraph whose function is the same as $\mathcal{F}_1(\beta)$ (though perhaps on a restricted domain), $\{H'\} \sqsubseteq S$ relative to $\mathcal{F}_1$.

If that first step is the application of some production $(\alpha \rightarrow D')$ for $D' \in A'$, we know there is some set $A_{D'} \subseteq A$ such that $\{D'\} \sqsubseteq A_{D'}$ relative to any mapping, and there is a production $(\alpha \rightarrow D)$ for every $D \in A_{D'}$. Apply each such production to the corresponding α in each member of R and collect the results in a set S. Since $\{G'\} \sqsubseteq R$ relative to $\mathcal{F}_1$ and since $\{D'\} \sqsubseteq A_{D'}$ relative to any mapping, $\{H'\} \sqsubseteq S$ relative to $\mathcal{F}_1$.

In both cases we have a set S such that $\{H'\} \sqsubseteq S$ relative to $\mathcal{F}_1$, and $H' \rightarrow^*_{Q'} I'$ in $n - 1$ steps. So by the inductive hypothesis S derives some terminal set T in Q for which $\{I'\} \sqsubseteq T$. □

Lemma 2.13 *Suppose we have two graphs G and G', possibly containing nonterminals, with $G' \sqsubseteq_{\text{weak}} G$ relative to $\mathcal{F}_1$; and suppose $G \Rightarrow^*_Q I$ for some terminal graph I with at least one execution. Then $G' \Rightarrow^*_{Q'} I'$ for some $I' \sqsubseteq_{\text{weak}} I$.*

Proof: by induction on the number n of steps in the derivation of I. In the base case $n = 0$, so G is terminal and $I = G$. G' must also be terminal. Since $G' \sqsubseteq_{\text{weak}} G$ relative to $\mathcal{F}_1$, and G' and G are fully terminal, $G' \sqsubseteq_{\text{weak}} G$.

In the inductive case $n > 0$: let H be the result of the first step in the derivation of some I with at least one execution, so that $G \Rightarrow_Q H \Rightarrow_Q^* I$. If the first step is the expansion of some vertex not corresponding to a vertex in G' then let $H' = G'$; since $H' \sqsubseteq_{\text{weak}} G$ relative to $\mathcal{F}_1$, $H' \sqsubseteq_{\text{weak}} H$ relative to $\mathcal{F}_1$.

If the first step is the expansion of some vertex corresponding to a vertex in G', it may or may not be the nonterminal α. If it isn't, we can expand the corresponding vertex in G' the same way, and get an H' for which $H' \sqsubseteq_{\text{weak}} H$ relative to $\mathcal{F}_1$.

If the first step is an expansion of α using some production $\alpha \rightarrow D$, note that since I has at least one execution, D must have an execution relative to some $\mathcal{F}$. It follows that D could not have been discarded in the thinning of the grammar: there must be some production $\alpha \rightarrow D'$ in Q' for which $D' \sqsubseteq_{\text{weak}} D$ relative to any $\mathcal{F}$. We can use that production on G' to get an H' for which $H' \sqsubseteq_{\text{weak}} H$ relative to $\mathcal{F}_1$.

In all cases we have a graph $H' \sqsubseteq_{\text{weak}} H$ relative to $\mathcal{F}_1$, and H derives some terminal graph I with at least one execution, in $n - 1$ steps. So, by the inductive hypothesis, $H' \Rightarrow_{Q'}^* I'$ for some $I' \sqsubseteq_{\text{weak}} I$. □

Theorem 2.14 $Q' \sqsubseteq Q$.

Proof: For the start graph S we have $\{S\} \sqsubseteq \{S\}$ relative to $\mathcal{F}_1$, so by Lemma 2.12 we can conclude that for any G' generated by Q' there is a set T generated by Q for which $\{G'\} \sqsubseteq T$. By Lemma 2.13 we can conclude that for any G generated by Q with at least one execution, there is a G' generated by Q' with $G' \sqsubseteq_{\text{weak}} G$.

These two conditions imply $Q' \sqsubseteq Q$: every graph generated by Q is thinner than or isomorphic to some graph generated by Q', and the two generated sets have matching executions. $\square$

2.5.3 Weak conservative thinning of right-hand sides

For weak conservative thinning (which is Thinner's approach) we want to be able to gather the right-hand sides of a nonterminal into a set (which will be executable, since the grammar is normal), thin that set using weak conservative thinning, and conclude that we have a weak-conservatively thinner grammar.

In the following discussion, let $Q = (N, T, P, S)$ be a normal trace grammar. For some nonterminal $\alpha \in N$ let A be the set $\{D \mid (\alpha \rightarrow D) \in P\}$, and suppose there is some executable set A' such that $A' \sqsubseteq A$ relative to every mapping $\mathcal{F}$ from nonterminals to *total* functions. Let Q' be a trace grammar formed from Q by removing the old productions for α and adding a new production $(\alpha \rightarrow D')$ for every $D' \in A'$. Because Q is normal each nonterminal corresponds to a partial function, obtained as the executable set generated from that nonterminal under Q. Define $\mathcal{F}_1$ to be this mapping from nonterminals to their actual functions under Q.

Lemma 2.15 *Suppose we have a graph G' and set R, possibly containing nonterminals, for which $\{G'\} \sqsubseteq_{\text{weak}} R$ relative to $\mathcal{F}_1$, and suppose G' derives some terminal graph I' in Q'. Then R derives some terminal set T in Q for which $\{I'\} \sqsubseteq_{\text{weak}} T$.*

Proof: similar to the proof of Lemma 2.12, but making use of the observation that if we have two sets $R' \sqsubseteq R$ relative to any mapping $\mathcal{F}$ to total functions, then $R' \sqsubseteq_{\text{weak}} R$ relative to any mapping $\mathcal{F}$ to partial functions. $\square$

Lemma 2.16 *Suppose we have two graphs G and G', possibly containing nonterminals, with $G' \sqsubseteq_{\text{weak}} G$ relative to $\mathcal{F}_1$; and suppose $G \Rightarrow_Q^* I$ for some terminal graph I with at least one execution. Then $G' \Rightarrow_{Q'}^* I'$ for some $I' \sqsubseteq_{\text{weak}} I$.*

Proof: This is the same as Lemma 2.13, and the same proof goes through in this case. □

Theorem 2.17 $Q' \sqsubseteq_{\text{weak}} Q$.

Proof: For the start graph S we have $\{S\} \sqsubseteq_{\text{weak}} \{S\}$ relative to $\mathcal{F}_1$, so by Lemma 2.15 we can conclude that for any G' generated by Q' there is a set T generated by Q for which $\{G'\} \sqsubseteq_{\text{weak}} T$. By Lemma 2.16 we can conclude that for any G generated by Q with at least one execution, there is a G' generated by Q' with $G' \sqsubseteq_{\text{weak}} G$. These two conditions imply $Q' \sqsubseteq_{\text{weak}} Q$: every graph generated by Q is thinner than or isomorphic to some graph generated by Q', and any execution in the set generated by Q corresponds to some execution in the set generated by Q'. □

Chapter 3

Thinner's Language

3.1 Overview

In the previous chapter we examined the trace grammar representation for first-order functional programs. Thinner operates on these trace grammars, finding and repairing violations of the PLC. But where do programs represented as trace grammars come from? I am certainly not proposing trace grammars as a programming language, so Thinner will have to start with programs written in a more customary language, compile them into trace grammars, optimize them, and then, if necessary, decompile them back into the source language.

The source language for Thinner is a small Lisp-like language called TL (naturally, Thinner's Language). TL satisfies two important restrictions: it is a first-order, purely functional language, and no errors can occur at runtime. I developed TL specifically for the Thinner project. It is a strongly typed language just large enough to express interesting first-order recursive functions over several data types. This chapter spells out the syntax and semantics of TL.

The chapter overview presents an informal description, and later sections of the chapter present a formal syntax and a denotational semantics. It is no great feat to give a denotational semantics for a simple, purely functional language like TL, and the reader may wonder why this material was not omitted, or at least relegated to an appendix. There are two points of interest:

- The semantics is an unusual one, since it gives a trace graph as the denotation of each expression. This clarifies the connection between TL and the trace grammars introduced in the previous chapter. In fact, the structure of the part of Thinner that compiles TL into trace grammars reflects the denotational semantics.

- The semantics is presented in a schematic form, with two semantic functions initially undefined. One definition for these functions yields a standard semantics for TL. An alternative definition developed in Chapter 4 places the relational constraint inference method in the framework of abstract interpretation.

3.1.1 Informal description of TL

In many ways TL programs are like pure first-order Lisp programs. The only available constructs are `defun`, `let`, `multiple-value-bind`, `values`, `if`, and function application.

The language is strongly typed. Every expression has a type and every function has a tuple of input types and a tuple of output types. The type names are `integer`, `boolean`, `bitvector`, and the three flat list types `(listof integer)`, `(listof boolean)` and `(listof bitvector)`. The `bitvector` type is a 32-bit word.

The boolean constants are given by the identifiers `#t` and `#f`. Neither these nor any other syntactic keyword may be used as the name of a user-defined function. Bitvector constants use the syntax `#h`*dddddddd*, where each *d* is a hexadecimal digit.

A `defun` may start with a declaration, using the syntax `(declare (`*typename identifier*`) ...)`, where each *identifier* is a parameter of the function. Thinner performs type inference while compiling, so many declarations may be omitted. Aside from the declaration, the body of a `defun` must be a single expression.

Table 3.1 shows the primitive functions of TL with their types. Several primitive functions are overloaded: for example, `equal` applies to any two objects of the same type. This overloading is resolved at compile time.

There are three typed empty list constants: the empty integer list `inil`, the empty boolean list `bnil`, and the empty bitvector list `vnil`. The `cdr` of an empty list is that empty list. The `car` of an empty list is always an element of the base type, even when the list is empty.[1]

3.2 Concrete syntax

Table 3.2 defines the syntax of TL using an extended BNF. In this table, $\langle x \rangle^*$ signifies a sequence of zero or more occurrences of $\langle x \rangle$, and $\langle x \rangle^+$ signifies a sequence of one or more. TL has the same lexical structure as Lisp: tokens are delimited by white space or by parentheses.

[1]Specifically, `(car inil)` is `0`, `(car bnil)` is `#f`, and `(car vnil)` is `#00000000`.

Operator Names	Operator Types
+, *	(integer × integer) → integer
-	(integer × integer) → integer integer → integer
abs	integer → integer
<, <=, >, >=	(integer × integer) → boolean
and, or	(boolean × boolean) → boolean (bitvector × bitvector) → bitvector
not	boolean → boolean bitvector → bitvector
cons	(integer × (listof integer)) → (listof integer) (boolean × (listof boolean)) → (listof boolean) (bitvector × (listof bitvector)) → (listof bitvector)
append	((listof integer) × (listof integer)) → (listof integer) ((listof boolean) × (listof boolean)) → (listof boolean) ((listof bitvector) × (listof bitvector)) → (listof bitvector)
cdr	(listof integer) → (listof integer) (listof boolean) → (listof boolean) (listof bitvector) → (listof bitvector)
length	(listof integer) → integer (listof boolean) → integer (listof bitvector) → integer
car	(listof integer) → integer (listof boolean) → boolean (listof bitvector) → bitvector
equal	(integer × integer) → boolean (boolean × boolean) → boolean (bitvector × bitvector) → boolean ((listof integer) × (listof integer)) → boolean ((listof boolean) × (listof boolean)) → boolean ((listof bitvector) × (listof bitvector)) → boolean

Table 3.1: Primitive functions and their types

$$
\begin{array}{rcl}
\langle \textit{program} \rangle & \rightarrow & \langle \textit{defun} \rangle^{+} \\
\langle \textit{defun} \rangle & \rightarrow & \texttt{(defun } \langle \textit{identifier} \rangle \ \texttt{(}\langle \textit{identifier} \rangle^{*}\texttt{)} \\
 & & \qquad \langle \textit{declare} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
 & & \quad \mid \texttt{(defun } \langle \textit{identifier} \rangle \ \texttt{(}\langle \textit{identifier} \rangle^{*}\texttt{)} \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{declare} \rangle & \rightarrow & \texttt{(declare } \langle \textit{declaration} \rangle^{+}\texttt{)} \\
\langle \textit{declaration} \rangle & \rightarrow & \texttt{(}\langle \textit{type} \rangle \ \langle \textit{identifier} \rangle\texttt{)} \\
\langle \textit{type} \rangle & \rightarrow & \langle \textit{basetype} \rangle \mid \texttt{(listof } \langle \textit{basetype} \rangle\texttt{)} \\
\langle \textit{basetype} \rangle & \rightarrow & \texttt{integer} \mid \texttt{boolean} \mid \texttt{bitvector} \\
\langle \textit{expression} \rangle & \rightarrow & \langle \textit{identifier} \rangle \mid \langle \textit{constant} \rangle \mid \langle \textit{conditional} \rangle \\
 & & \quad \mid \langle \textit{multiple value bind} \rangle \mid \langle \textit{let} \rangle \mid \langle \textit{values} \rangle \\
 & & \quad \mid \langle \textit{terminal call} \rangle \mid \langle \textit{nonterminal call} \rangle \\
\langle \textit{constant} \rangle & \rightarrow & \langle \textit{number} \rangle \mid \langle \textit{boolean} \rangle \mid \langle \textit{bitvector} \rangle \mid \texttt{inil} \mid \texttt{bnil} \mid \texttt{vnil} \\
 & & \quad \mid \texttt{'(}\langle \textit{number} \rangle^{+}\texttt{)} \mid \texttt{'(}\langle \textit{boolean} \rangle^{+}\texttt{)} \mid \texttt{'(}\langle \textit{bitvector} \rangle^{+}\texttt{)} \\
\langle \textit{boolean} \rangle & \rightarrow & \texttt{\#t} \mid \texttt{\#f} \\
\langle \textit{bitvector} \rangle & \rightarrow & \texttt{\#h}\langle \textit{hex digit} \rangle^{32} \\
\langle \textit{conditional} \rangle & \rightarrow & \texttt{(if } \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{multiple value bind} \rangle & \rightarrow & \texttt{(multiple-value-bind } \texttt{(}\langle \textit{identifier} \rangle^{+}\texttt{)} \\
 & & \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{let} \rangle & \rightarrow & \texttt{(let } \texttt{(}\langle \textit{binding} \rangle^{*}\texttt{)} \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{binding} \rangle & \rightarrow & \texttt{(}\langle \textit{identifier} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{values} \rangle & \rightarrow & \texttt{(values } \langle \textit{expression} \rangle^{+}\texttt{)} \\
\langle \textit{terminal call} \rangle & \rightarrow & \texttt{(}\langle \textit{binary terminal} \rangle \ \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
 & & \quad \mid \texttt{(}\langle \textit{unary terminal} \rangle \ \langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{binary terminal} \rangle & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \\
 & & \quad \mid \texttt{and} \mid \texttt{or} \mid \texttt{cdr} \mid \texttt{cons} \mid \texttt{append} \mid \texttt{equal} \\
\langle \textit{unary terminal} \rangle & \rightarrow & \texttt{-} \mid \texttt{abs} \mid \texttt{not} \mid \texttt{length} \mid \texttt{car} \\
\langle \textit{nonterminal call} \rangle & \rightarrow & \texttt{(}\langle \textit{identifier} \rangle \ \langle \textit{expression} \rangle^{*}\texttt{)}
\end{array}
$$

Table 3.2: Extended BNF grammar for TL

3.3 Trace graph constructors

Recall that we defined a trace grammar as a 4-tuple (N, T, P, S), where N was a set of "nonterminal labels" and T a set of "terminal labels". Labels of both kinds occurred in vertices, in the graphs that made up the right-hand sides of productions in P. The nonterminal labels also occurred as the left-hand sides of productions, and this is how nonterminal vertices were connected with their productions.

To give a semantics for TL we must be a bit more specific about the nature of these labels. In the semantics that follows, each vertex label will be either:

- a relation, for terminal vertices;
- an identifier, for vertices corresponding to nonterminal function calls; or
- a pair of trace graphs (giving the right-hand sides of two productions immediately), for vertices corresponding to conditionals.

To build up these trace graphs denotationally we will use several trace graph constructor functions. These are defined below. (In these definitions, and throughout this chapter, boldface variables denote sequences.)

Identity(n) is an n-input, n-output graph that passes each input directly to the corresponding output. Figure 3.1 gives an example.

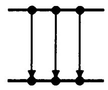
Identity(3) = 

Figure 3.1: A trace graph constructed by Identity

Constant$(\boldsymbol{i}, v)$ is a graph depending on the constant value v and on $|\boldsymbol{i}|$. (The sequence $\boldsymbol{i}$ will be a sequence of identifiers, but only the length matters.) The graph has $|\boldsymbol{i}|$ inputs, which are all unused; it has no vertices other than V_i and V_o; and it has one output, whose source is the constant v. Figure 3.2 gives an example.

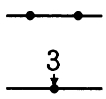
$\text{Constant}((\mathtt{x}, \mathtt{y}), 3) =$ 

Figure 3.2: A trace graph constructed by Constant

Select$(\boldsymbol{i}, j)$ is a graph depending on a sequence of identifiers $\boldsymbol{i}$ and an identifier j which must occur in $\boldsymbol{i}$. The graph has $|\boldsymbol{i}|$ inputs and one output; it has no vertices other than V_i and V_o; and the source for the output is the first input m for which $\boldsymbol{i}_m = j$. Figure 3.3 gives an example.

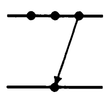
$\text{Select}((\mathsf{a}, \mathsf{b}, \mathsf{c}), \mathsf{c}) =$ 

Figure 3.3: A trace graph constructed by Select

Single(m, n, l) is an m-input, n-output graph with a single m-input, n-output vertex v with label$(v) = l$. The inputs of the vertex are the graph inputs in order, and the outputs of the vertex are the graph outputs in order. Figure 3.4 gives an example.

Cat$(G_1, \ldots, G_n)$ combines $n \geq 1$ graphs into a new graph H, whose input arity is the sum of the input arities of the G_i's and whose output arity is the sum of the output arities of the G_i's. Each G_i is duplicated in H (not including the input and output vertices), and is connected in order to the inputs and outputs of H. Figure 3.5 gives an example.

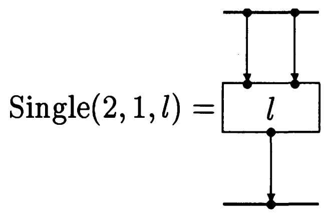


Figure 3.4: A trace graph constructed by Single

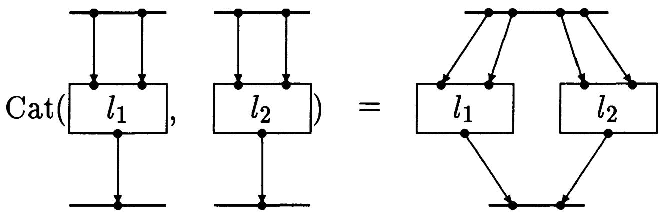


Figure 3.5: A trace graph constructed by Cat

Meld$(G_1, \ldots, G_n)$ combines $n \geq 2$ graphs into a new graph H as follows. The first $n-1$ graphs must all have the same input arity k; this will also be the input arity of H. Each G_i for $i < n$ is duplicated in H (not including the input and output vertices), with the inputs connected to the corresponding inputs of H. The sum of the output arities of the first $n-1$ graphs is the input arity of G_n; the outputs of the first $n-1$ graphs are connected to the inputs of G_n in order. The outputs of H are the outputs of G_n. Figure 3.6 gives an example.

The six functions Identity, Constant, Select, Single, Cat and Meld are used in the denotational semantics that follows, to construct the trace graphs which are the denotations of expressions.

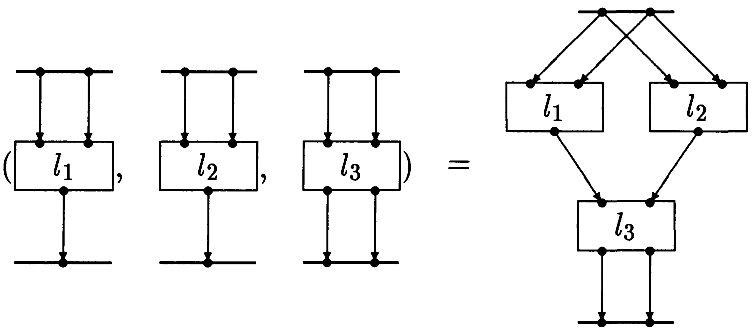
Meld(

Figure 3.6: A trace graph constructed by Meld

3.4 Types and type inference

As noted above, TL is strongly typed. Its types are `integer`, `boolean`, `bitvector`, and the three flat list types `(listof integer)`, `(listof boolean)` and `(listof bitvector)`. All type checking is done at compile time.

We will not labor through a type semantics here. Instead, we will treat type checking as a syntactic preprocessing step. We'll assume that a consistent type has been determined for every expression, all overloaded operators have been disambiguated, and the input tuples and output tuples of all functions have been typed. Where necessary, we'll refer to the derived type of an expression or function in the semantics using type subscripts.

3.5 Abstract syntax

These are the abstract syntactic categories:

C ∈ Con	constants (including quoted lists)
I ∈ Ide	identifiers
T ∈ Ter	terminal functions

$\mathrm{E} \in \mathrm{Exp}$ expressions

$\mathrm{D} \in \mathrm{Def}$ defuns

$\mathrm{P} \in \mathrm{Pro}$ programs

An abstract syntax for TL is as follows. (Remember that we're using bold type for sequences.)

$\mathrm{E} \rightarrow \mathrm{C} \mid \mathrm{I}' \mid$ `(`T $\mathbf{E}$`)` $\mid$ `(`I' $\mathbf{E}$`)`

$\mid$ `(if` E E′ E″`)`

$\mid$ `(values` $\mathbf{E}$`)`

$\mid$ `(let ((`$\mathbf{I}'_1$ $\mathbf{E}'_1$`) ... (`$\mathbf{I}'_n$ $\mathbf{E}'_n$`))` E`)`

$\mid$ `(multiple-value-bind (`$\mathbf{I}'$`)` E E′`)`

$\mathrm{D} \rightarrow$ `(defun` I′ `(`$\mathbf{I}$`)` E`)`

$\mathrm{P} \rightarrow \mathbf{D}$

3.6 Semantic sets

It is customary to use domains for the denotations of programming constructs, but we will use sets instead. This simplification is possible because we will not be using recursive domain definitions or fixed points, and we will not need a bottom element to represent non-termination.[2]

The denotations will be elements of these semantic sets:

V values (booleans, numbers, bitvectors, and lists thereof)

C relations on V

[2]Actually, there is a hidden recursiveness in the semantic sets: the label of a vertex in a trace graph may be a pair of trace graphs. But since the whole structure of a trace grammar is finite, this can easily be accomodated; and I didn't want to introduce a lot of unnecessary mathematical machinery.

Z	trace graphs
X	pairs (G,f) where f is an execution of $G \in Z$
N	nonterminals (i, G) for identifier $i \in \mathrm{I}$ and $G \in Z$

Note that the syntactic category I of identifiers persists in the denotations of nonterminals.

The semantic functions are as follows:

$$\mathcal{K} : \mathrm{Con} \to V$$

$$\mathcal{E} : \mathrm{Exp} \to \mathrm{Ide}^* \to Z$$

$$\mathcal{D} : \mathrm{Def} \to N$$

$$\mathcal{Q} : \mathrm{Pro} \to \mathrm{Exp} \to \mathcal{P}(X)$$

$$\mathcal{T} : \mathrm{Ter} \to C$$

(Here, $\mathcal{P}(S)$ denotes the set of subsets of S. We will also use $\mathcal{P}_f(S)$ to denote the set of *finite* subsets of S.)

The semantic function $\mathcal{K}$ maps each constant to the corresponding value. $\mathcal{E}$ maps each expression and list of identifiers to a trace graph. The list of identifiers will determine the number of inputs to the graph, and will guide the connections for identifier references within the expression. $\mathcal{D}$ maps each function definition to a nonterminal, which is just an identifier (the name of the function) and a trace graph (the right-hand side of the single production for that function).

In the interactive style of Lisp-like languages, a program is a collection of function definitions that define an environment in which expressions may be evaluated. Semantic function $\mathcal{Q}$ reflects this idea: it maps each program to function that maps an expression (which is assumed to contain no free variables) to a set of executed trace graphs. (In the standard semantics that set will either be of size zero, if the evaluation of the expression given the definitions of the program does not terminate, or of size one, if it does.)

$\mathcal{T}$ determines the constraint associated with each terminal function. We will also use an auxilliary function Lang; this is the function that determines the language of executed trace graphs generated by a grammar.

$$\text{Lang} : Z \to \mathcal{P}_f(N) \to \mathcal{P}(X)$$

In Section 3.8, below, we will define $\mathcal{T}$ and Lang in a way that yields a standard semantics for TL. We will try alternate definitions in Chapter 4, when we place the relational constraint method of inference in the context of abstract interpretation.

3.7 Semantic schema

The definition of the semantic function $\mathcal{K}$ mapping syntactic constants to values is omitted.

$$
\begin{aligned}
&\mathcal{E}[\![\mathrm{C}]\!]\mathbf{I} = \mathrm{Constant}(\mathbf{I}, \mathcal{K}(\mathrm{C}))\\
&\mathcal{E}[\![\mathrm{I}']\!]\mathbf{I} = \mathrm{Select}(\mathbf{I}, \mathrm{I}')\\
&\mathcal{E}[\![\texttt{(}\mathrm{T}_{\boldsymbol{\alpha}\to\boldsymbol{\beta}}\ \mathbf{E}\texttt{)}]\!]\mathbf{I} = \mathrm{Meld}(\mathcal{E}[\![\mathrm{E}_1]\!]\mathbf{I}, \ldots, \mathcal{E}[\![\mathrm{E}_{|\boldsymbol{\alpha}|}]\!]\mathbf{I}, \mathrm{Single}(|\boldsymbol{\alpha}|, |\boldsymbol{\beta}|, \mathcal{T}(\mathrm{T})))\\
&\mathcal{E}[\![\texttt{(}\mathrm{I}'_{\boldsymbol{\alpha}\to\boldsymbol{\beta}}\ \mathbf{E}\texttt{)}]\!]\mathbf{I} = \mathrm{Meld}(\mathcal{E}[\![\mathrm{E}_1]\!]\mathbf{I}, \ldots, \mathcal{E}[\![\mathrm{E}_{|\boldsymbol{\alpha}|}]\!]\mathbf{I}, \mathrm{Single}(|\boldsymbol{\alpha}|, |\boldsymbol{\beta}|, \mathrm{I}'))\\
&\mathcal{E}[\![\texttt{(if}\ \mathrm{E}\ \mathrm{E}'_{\boldsymbol{\alpha}}\ \mathrm{E}''_{\boldsymbol{\alpha}}\texttt{)}]\!]\mathbf{I} = \mathrm{Meld}(\mathrm{Identity}(|\mathbf{I}|), \mathcal{E}[\![\mathrm{E}]\!]\mathbf{I},\\
&\qquad\qquad \mathrm{Single}(|\mathbf{I}|+1, |\boldsymbol{\alpha}|,\\
&\qquad\qquad\qquad (\mathrm{Cat}(\mathcal{E}[\![\mathrm{E}']\!]\mathbf{I}, \mathrm{Single}(1, 0, (\mathrm{true}))),\\
&\qquad\qquad\qquad \mathrm{Cat}(\mathcal{E}[\![\mathrm{E}'']\!]\mathbf{I}, \mathrm{Single}(1, 0, (\mathrm{false}))))))\\
&\mathcal{E}[\![\texttt{(values}\ \mathbf{E}\texttt{)}_{\boldsymbol{\alpha}}]\!]\mathbf{I} = \mathrm{Meld}(\mathcal{E}[\![\mathrm{E}_1]\!]\mathbf{I}, \ldots, \mathcal{E}[\![\mathrm{E}_{|\boldsymbol{\alpha}|}]\!]\mathbf{I}, \mathrm{Identity}(|\boldsymbol{\alpha}|))\\
&\mathcal{E}[\![\texttt{(let ((}\mathbf{I}'_1\ \mathbf{E}'_1\texttt{)}\ \ldots\ \texttt{(}\mathbf{I}'_n\ \mathbf{E}'_n\texttt{))}\ \mathrm{E}\texttt{)}]\!]\mathbf{I} = \mathrm{Meld}(\mathcal{E}[\![\mathbf{E}'_1]\!]\mathbf{I}, \ldots, \mathcal{E}[\![\mathbf{E}'_n]\!]\mathbf{I},\\
&\qquad\qquad \mathrm{Identity}(|\mathbf{I}|),\\
&\qquad\qquad \mathcal{E}[\![\mathrm{E}]\!](\mathbf{I}' \cdot \mathbf{I}))
\end{aligned}
$$

$$\mathcal{E}[\![\texttt{(multiple-value-bind (}\mathbf{I}'\texttt{) E E}'\texttt{)}]\!]\mathbf{I} = \mathrm{Meld}(\mathcal{E}[\![\mathrm{E}]\!]\mathbf{I},\ \mathrm{Identity}(|\mathbf{I}|),\ \mathcal{E}[\![\mathrm{E}']\!](\mathbf{I}'\cdot\mathbf{I}))$$

$$\mathcal{D}[\![\texttt{(defun I}'\texttt{ (I) E)}]\!] = (\mathrm{I}', \mathcal{E}[\![\mathrm{E}]\!]\mathbf{I})$$

$$\mathcal{Q}[\![\mathbf{D}]\!]\mathrm{E} = \mathrm{Lang}(\mathcal{E}[\![\mathrm{E}]\!](), \{\mathcal{D}[\![\mathbf{D}_1]\!], \ldots, \mathcal{D}[\![\mathbf{D}_n]\!]\})$$

3.8 Standard semantics

Now to define the remaining semantic function $\mathcal{T}$ and the auxiliary function Lang. To get a standard semantics we define $\mathcal{T}$ so that the meaning of each terminal function identifier is the relation normally associated with the corresponding function on values:

$$\mathcal{T}[\![\texttt{+}]\!] = \{(V_0, V_1, V_2) \mid V_2 = V_0 + V_1\}$$

$$\mathcal{T}[\![\texttt{not}_{\text{boolean}\to\text{boolean}}]\!] = \{(\text{true}, \text{false}), (\text{false}, \text{true})\}$$

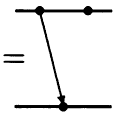
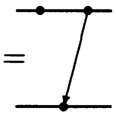
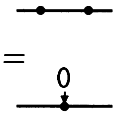
and so forth. We define Lang to be the function that generates the set of fully terminal trace graphs from a trace grammar in the usual way:

$$\mathrm{Lang}(G, S) = \{(H, f) \mid H \text{ is a fully terminal trace graph derivable from start graph } G \text{ in grammar } S\text{, and } f \text{ is an execution of } H\}$$

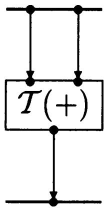
3.9 Example

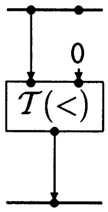
For example, consider the program `(defun q (a b) (if (< a 0) (+ a b) a))`. The simplest expressions in the program are the identifiers `a` and `b` and the constant

0. These expressions have the following denotations, constructed by Select and Constant:

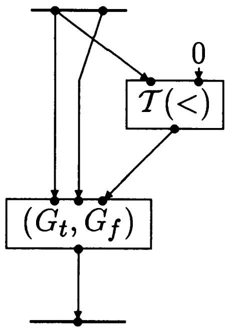
$\mathcal{E}[\![\mathrm{a}]\!](\mathrm{a},\mathrm{b}) =$  $\mathcal{E}[\![\mathrm{b}]\!](\mathrm{a},\mathrm{b}) =$  $\mathcal{E}[\![0]\!](\mathrm{a},\mathrm{b}) =$ 

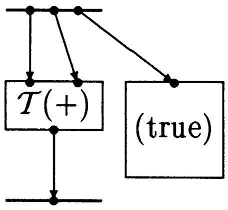
From these simple denotations, the meanings of the compound expressions `(+ a b)` and `(< a 0)` are constructed:

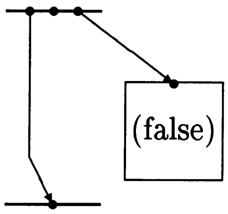
$\mathcal{E}[\![$`(+ a b)`$]\!](\mathrm{a},\mathrm{b}) =$ 

$\mathcal{E}[\![$`(< a 0)`$]\!](\mathrm{a},\mathrm{b}) =$ 

Now we can construct the denotation of `(if (< a 0) (+ a b) a)`. This makes use of a nonterminal vertex whose label is a pair of graphs.

$\mathcal{E}[\![$`(if (< a 0) (+ a b) a)`$]\!](\mathrm{a},\mathrm{b}) =$ 

$G_t =$ 

$G_f =$ 

Let G be this final graph, $G = \mathcal{E}[\![$`(if (< a 0) (+ a b) a)`$]\!](\mathrm{a},\mathrm{b})$. Then

$$\mathcal{D}[\![\texttt{(defun q (a b) (if (< a 0) (+ a b) a))}]\!] = (\mathrm{q}, G)$$

which is the pair giving G as the only production for q; and finally

$$\mathcal{Q}[\![\texttt{(defun q (a b) (if (< a 0) (+ a b) a))}]\!] = \lambda \mathrm{E}\,.\,\mathrm{Lang}(\mathcal{E}[\![\mathrm{E}]\!](), \{(\mathrm{q}, G)\})$$

Chapter 4

Finding Violations

4.1 Overview

How can violations of the PLC be identified? A trace grammar violates the PLC if there is a conservatively thinner grammar, and it has already been observed that even for individual trace graphs the question of whether a given thinning is conservative or not is undecidable. It follows that there is no way to automate an optimizing step that says, "Now find all violations of the PLC in the grammar."

At first glance this may seem like a major obstacle, but in fact undecidability is the least of our worries. In early prototypes of Thinner I experimented with several different "fast" inference techniques: a Prolog-style backward-chaining inference engine and a term-rewriting system, among others. It quickly became clear that the major difficulty in finding violations of the PLC is not undecidability but intractability. Undecidability only says that no algorithm can correctly detect all violations of the PLC. One can live with this kind of limitation in an optimizer; but I found many seemingly obvious violations of the PLC that no method I tried would detect reasonably quickly.

My solution to this problem was to build a semantic analyzer that combines traditional techniques, like the computation of normal forms of expressions, with a new method of inference called *relational constraint* [Web92b]. This semantic analyzer is the subject of this chapter.

Research in automated deduction has usually focused on generality and adequacy rather than efficiency [Bib91]. Compared with the state of the art in automated deduction, the relational constraint method is not very powerful. Its chief virtue is that it develops a collection of useful inferences about a trace graph of size n in $O(n^2)$ time. So it strikes a balance between power and efficiency—the right balance for a semantic analyzer like Thinner's.

4.1.1 The semantic analyzer

Another part of Thinner, described in Chapter 6, inspects the trace grammar being optimized and proposes individual trace graphs for semantic analysis. It is the job of the semantic analyzer to identify as many conservative thinnings of that graph as it can, as quickly as it can. This means finding equivalences: two vertex outputs that are assigned equal values in every execution, or an individual vertex output that is assigned the same (constant) value in every execution.

To do this the semantic analyzer makes use of two complementary techniques: the relational constraint method, and the computation and comparison of normal forms of expressions. These two methods complement each other well. Each finds equivalences that the other misses.

Relational constraint is described in Section 4.2. It is a fully automatic inference technique with a forward-chaining flavor: it doesn't prove or disprove queries, it reports observations. It starts with a fixed, finite vocabulary of binary relations; for each data type this vocabulary forms a complete lattice. Relational constraint focuses on strengthening relations in this lattice, developing the strongest correct statement it can about the relation between any two objects in its universe.

The output of the relational constraint algorithm is a binary relation for every pair of "objects" of the same type. These objects are:

- all vertex outputs and constants from the graph; and
- the integer constants `0` and `1`, the boolean constants `#t` and `#f`, the bitvector constant `00000000x0`, and the three typed empty lists—whether they occur in the graph or not.

The vocabulary of binary relations for a given data type is fixed. It forms a complete lattice in which the greatest lower bound of a set of relations yields a relation which is their conjunction, and the least upper bound of a set of relations yields a relation which is their disjunction. Relations are strengthened monotonically during inference: they start out at the top of the lattice (the relation which is true of all pairs) and may grow stronger during inference, but are never weakened or retracted.[1] The vocabulary of binay relations is described in detail in Section 4.2.1.

The leverage for strengthening relations comes from *constraints* on tuples of relations. A constraint (P, S) gives a tuple P of pairs of objects and a specification S of allowable tuples of relations on those pairs. Constraints come in four flavors:

Functional constraints. Each operator in a program restricts the relations among its input and output values. In some cases (like the absolute value function) this restriction is a simple input-output relation which can be asserted immediately. In other cases (like subtraction) the relations among the

[1] From a certain point of view the lattice of relations appears upside-down, since it is natural to think that information increases as you move up in a lattice. Not so in this case: information increases downward. The reason for this will become more clear in the next section, when we observe that these relations are built from sets in such a way that the disjunction of relations ($\vee$), the union of the sets ($\cup$) and least upper bound in the lattice ($\sqcup$) coincide.

input and output values, and relations with certain constants, are restricted in a more complex way. For example, if we have three objects a, b and c with $a =$ `(- b c)`, and if $b = c$, a functional constraint excludes the possibility that $a \neq 0$.

Transitivity constraints. For any three objects a, b and c of the same type, the relations among a, b and c are restricted by transitivity. For example, given that $a < b$ and $b < c$, the transitivity constraint for a, b and c excludes the possibility that $a \geq c$. (Thinner limits its use of transitivity constraints to certain interesting triples.)

List property constraints. List property constraints ensure that for each pair of list objects (a, b) the relation between a and b, the relation between car(a) and car(b), and the relation between length(a) and length(b), are as strong as possible taken jointly. For example, if we have two list objects a and b and two base-type objects c and d, and if we know that $c = \text{car}(a)$, $d = \text{car}(b)$, and $a = b$, then the list property constraint for a and b excludes the possibility that $c \neq d$.

Transparency constraints. For each pair of calls to the same function, a transparency constraint insists that the relations reflect referential transparency. For example, if the program contains both $a = f(x)$ and $b = f(y)$, and if $x = y$, a transparency constraint excludes the possibility that $a \neq b$.

The nature of the specification of allowable tuples S in a constraint (P, S) and the method by which S is used to strengthen the relations on pairs in P are described in Section 4.2.2. The full relational constraint algorithm is presented and analyzed in Section 4.2.3.

The other part of Thinner's semantic analysis is the computation and comparison of normal forms. A normal form for the expression computing each object is recorded in a hash table, and when two normal forms are found to be identical, the objects must be equal. This is a standard technique and doesn't require much explanation; for completeness, Section 4.3 outlines the normal forms used by Thinner.

4.2 Relational constraint

4.2.1 The vocabulary of binary relations

When other parts of Thinner propose a trace graph for semantic analysis, the analyzer's job is to find equivalences on objects in that graph. In fact, the relational constraint method finds a binary relation for every pair of objects in the graph; some of these may imply equivalence, but even those that do not provide leverage for relevant inference.

We begin by defining a small set X^t of *primitive* binary relations for each data type t. The primitive relations for a type are exhaustive and mutually exclusive, so that for any constants a and b of type t, there is exactly one $r \in X^t$ for which arb. Further, every primitive relation has a reversed version, so that if $q \in X^t$, there is some $r \in X^t$ such that arb if and only if bqa. (q may be symmetric, in which case r and q will be identical.)

The primitive relations are used to build a lattice of *expressible* relations, which makes up the semantic analyzer's vocabulary. We'll denote an expressible relation by indicating a set R of primitive relations: we define xRy as $\bigvee_{r \in R} xry$. So an expressible relation is just a set of primitive relations, one of which must be satisfied;

each subset of the set of primitive relations gives a different expressible relation.

Because the primitive relations are exhaustive and mutually exclusive, this construction yields a lattice of expressible relations in which the disjunction of two relations Q and R is the least upper bound $(Q \cup R)$ and their conjunction is the greatest lower bound $(Q \cap R)$. The set X^t of all primitives for type t is the expressible relation which is true of all pairs of objects of type t, and the empty set is the expressible relation which is never true; we'll denote these relations $\top$ and $\bot$, respectively. For any expressible relation Q there is a relation R with the property that xQy if and only if $\neg(xRy)$; R is simply $X^t \setminus Q$. Finally, for any expressible relation Q there is an expressible relation R with the property that xQy if and only if yRx; R can be constructed by simply reversing every primitive in Q. (Again, Q may be symmetric, in which case Q and R will be identical.)

By construction, this vocabulary of expressible relations has the following properties:

- If two relations are expressible, their conjunction and disjunction are expressible.
- If a relation is expressible, so is its negation.
- If a relation is expressible, the same relation with arguments reversed is also expressible.

There is an obvious bit-coding of the expressible relations using one bit for each primitive relation in X^t. Using this representation disjunction can be computed using logical-or, conjunction using logical-and, and negation using logical-not.

The method of relational constraint will work for any vocabulary with the lattice properties described above. The remainder of this section describes the vocabulary used by Thinner.

Integers

Thinner's primitive relations on integers a and b describe the sign of a: $a < 0$, $a = 0$, or $a > 0$; the sign of b: $b < 0$, $b = 0$, or $b > 0$; and the difference of the magnitudes $|a| - |b|$: $|a| - |b| \leq -2$, $|a| - |b| = -1$, $|a| - |b| = 0$, $|a| - |b| = 1$, or $|a| - |b| \geq 2$. Not all combinations of these properties are consistent—for example, if both a and b are 0 the difference of their magnitudes can only be 0. But 29 of the combinations are; and these 29 make up the exhaustive set X^{int} of mutually exclusive relations on integers. Table 4.1 shows the notation we'll use for these primitive relations.

	Difference of magnitudes, $\lvert a\rvert - \lvert b\rvert$:				
	≤ -2	$= -1$	$= 0$	$= 1$	≥ 2
$a < 0,\ b < 0$	--<2+	--<1	--=0	-->1	-->2+
$a < 0,\ b = 0$				-0>1	-0>2+
$a < 0,\ b > 0$	-+<2+	-+<1	-+=0	-+>1	-+>2+
$a = 0,\ b < 0$	0-<2+	0-<1			
$a = 0,\ b = 0$			00=0		
$a = 0,\ b > 0$	0+<2+	0+<1			
$a > 0,\ b < 0$	+-<2+	+-<1	+-=0	+->1	+->2+
$a > 0,\ b = 0$				+0>1	+0>2+
$a > 0,\ b > 0$	++<2+	++<1	++=0	++>1	++>2+

Table 4.1: Notation for the primitive relations in X^{int}

There are 2^{29} subsets of X^{int}, giving the same number of expressible relations on integers. A variety of well-known relations on integers are among the expressible; Table 4.2 gives some examples.

xRy, for $R =$	Description
`{--=0, 00=0, ++=0}`	$x = y$
`{-->2+, -->1, -0>2+, -0>1, 0+<2+, 0+<1, ++<2+, ++<1}`	$x < y$
`{--<1, 0-<1, +0>1, ++>1}`	$x = \mathrm{succ}(y)$
`{--=0, -+=0, 00=0, +-=0, ++=0}`	$\lvert x \rvert = \lvert y \rvert$

Table 4.2: Examples of common expressible relations on integers

Booleans

There are four possible combinations of two booleans x and y. This yields a set X^{bool} of four primitive relations, which are denoted `ff`, `ft`, `tf`, and `tt`, with $x\{\texttt{ft}\}y$ if and only if $x =$ false and $y =$ true, and so on.

There are sixteen subsets of the X^{bool}, giving the same number of expressible relations on booleans. These are described in Table 4.3.

xRy, for $R =$	Description
`{}`	false for all x and y
`{ff}`	$x =$ false and $y =$ false
`{ft}`	$x =$ false and $y =$ true
`{tf}`	$x =$ true and $y =$ false
`{tt}`	$x =$ true and $y =$ true
`{ff, ft}`	$x =$ false
`{ff, tf}`	$y =$ false
`{ff, tt}`	$x = y$
`{ft, tf}`	$x = \overline{y}$
`{ft, tt}`	$y =$ true
`{tf, tt}`	$x =$ true
`{ft, tf, tt}`	$\overline{y} \rightarrow x$
`{ff, tf, tt}`	$y \rightarrow x$
`{ff, ft, tt}`	$x \rightarrow y$
`{ff, ft, tf}`	$x \rightarrow \overline{y}$
`{ff, ft, tf, tt}`	true for all x and y

Table 4.3: The expressible relations on booleans

Bitvectors

There are 15 nonempty sets of ordered pairs over the universe $\{0,1\}$. Each primitive relation r on bitvectors can be described by one of these sets: if A is such a set, the corresponding primitive relation r_A has xr_Ay if and only if for every bit position, the ordered pair of bits drawn from x and y at that position is in A, and every pair in A occurs as the pair of bits drawn at some position. In other words:

$$xr_Ay \equiv \left(\bigwedge_{n\in[0..31]} (x_n, y_n) \in A\right) \bigwedge \left(\bigwedge_{(a,b)\in A} \left(\bigvee_{n\in[0..31]} (a,b) = (x_n, y_n)\right)\right)$$

The set X^{bitv} of primitive relations on bitvectors is the set of the fifteen such relations r_A. These primitive relations are exhaustive and mutually exclusive, as required, and a primitive relation r_A can be reversed by reversing all the ordered pairs in A. We'll use the notation shown in Table 4.4 for these primitive relations, but remember that these relations are mutually exclusive. The relation `v=` is like equality, but by definition it excludes `v00` (where both are `00000000x0`) and `v11` (where both are `FFFFFFFFx0`).

There are 2^{15} subsets of X^{bitv}, giving the same number of expressible relations on bitvectors. Examples of expressible relations on bitvectors are given in Table 4.5.

Lists

There are actually three list types, but the primitive and expressible relations are the same for all three. These relations should be considered as applying only to lists of matching type.

Thinner's primitive relations on lists a and b describe whether the shorter is a suffix of the longer; whether the two lists have the same car; and the difference of the lengths $|a| - |b|$: $|a| - |b| \leq -2$, $|a| - |b| = -1$, $|a| - |b| = 0$, $|a| - |b| = 1$, or

Notation	ar_Ab, for $A =$
v00	$\{(0,0)\}$
v01	$\{(0,1)\}$
v10	$\{(1,0)\}$
v11	$\{(1,1)\}$
v0x	$\{(0,0),(0,1)\}$
vx0	$\{(0,0),(1,0)\}$
v=	$\{(0,0),(1,1)\}$
v=-	$\{(0,1),(1,0)\}$
vx1	$\{(0,1),(1,1)\}$
v1x	$\{(1,0),(1,1)\}$
v>-	$\{(0,1),(1,0),(1,1)\}$
v>	$\{(0,0),(1,0),(1,1)\}$
v<	$\{(0,0),(0,1),(1,1)\}$
v<-	$\{(0,0),(0,1),(1,0)\}$
vxx	$\{(0,0),(0,1),(1,0),(1,1)\}$

Table 4.4: The set X^{bitv} of primitive relations on bitvectors

xRy, for $R =$	Description
{v00}	$x = y = 0$
{v00, v11, v=}	$x = y$
{v01, v10, v=-}	$x = \overline{y}$
{v11, v=}	$x = y \neq 0$
{v00, v01, v11, v0x, v=, vx1, v<}	$x \vee y = y$

Table 4.5: Some common expressible relations on bitvectors

$|a| - |b| \geq 2$. Not all combinations of these properties are consistent: if a and b have the same length and are suffixes of each other, they must have the same car. But 19 of the combinations are; and these 19 make up the exhaustive set X^{list} of mutually exclusive relations on lists. Table 4.6 shows the notation we'll use for these primitive relations.

	$\lvert a\rvert - \lvert b\rvert \leq -2$	$\lvert a\rvert - \lvert b\rvert = -1$	$\lvert a\rvert = \lvert b\rvert$	$\lvert a\rvert - \lvert b\rvert = 1$	$\lvert a\rvert - \lvert b\rvert \geq 2$
suffix	s<2+	s<1		s>1	s>2+
same car	c<2+	c<1	c=0	c>1	c>2+
both	sc<2+	sc<1	sc=0	sc>1	sc>2+
neither	<2+	<1	=0	>1	>2+

Table 4.6: The primitive relations in X^{list}

There are 2^{19} subsets of X^{list}, giving the same number of expressible relations on lists. Examples are given in Table 4.7, below.

xRy, for $R =$	Description
{s<1, sc<1}	$x = \text{cdr}(y)$ (y not empty)
{sc=0, c=0, =0}	$\lvert x\rvert = \lvert y\rvert$
$X^{\text{list}} \setminus$ {sc=0}	$x \neq y$

Table 4.7: Some common expressible relations on lists

4.2.2 Finding strong relations

In the previous section we established a vocabulary of expressible relations, giving a complete lattice of relations on each type. Of course, every pair of identically typed objects in a trace graph satisfies the top relation for that type, but our goal is to find the strongest relation we can, as quickly as possible.

The table of relations

The relational constraint algorithm uses a table of relations giving the strongest relation currently known for each pair of objects. This table is always relationally symmetric: the relation in the table for (a, b) is always the reverse of the relation for (b, a).

Most entries in the table of relations are initially the top relation for their type, indicating that Thinner knows nothing about the relation for that pair. But there are several exceptions:

- The relation of an object with itself is initially the equality relation. Note that the equality relation need not be maximally strong: for example, the equality relation on integers is $\{$`--=0`, `00=0`, `++=0`$\}$, but if the semantic analyzer discovers that the object is in fact positive, this relation can be strengthened to $\{$`++=0`$\}$.

- The relation of an object with a constant can be initialized to something stronger than the top relation. For example, the relation between a list object a and the empty list can be initialized to $\{$`s>2+`, `sc>2+`, `s>1`, `sc>1`, `sc=0`$\}$.

- The relation between two constants can be computed immediately. Exactly one primitive relation applies to any ordered pair of constants.

- Some pairs have an initial relation established by a "single-relation operator" as described below.

Single-relation operators

Some of the operators in TL establish a relation on a single pair of objects. For example, suppose a trace graph includes the computation $a =$ `(-` b`)`. This establishes a relation between a and b that is expressible in the language of relations, so all we need do is strengthen the original relation between a and b, $\top$, to the negation relation, $\{\texttt{+-=0}, \texttt{-+=0}, \texttt{00=0}\}$. Table 4.8 shows the other TL operators that can be treated this way.

Operation	Relation on (a, b)
$a =$ `(- b)`	$\{\texttt{+-=0}, \texttt{-+=0}, \texttt{00=0}\}$
$a =$ `(abs b)`	$\{\texttt{++=0}, \texttt{-+=0}, \texttt{00=0}\}$
$a =$ `(not b)`, a and b boolean	$\{\texttt{ft}, \texttt{tf}\}$
$a =$ `(not b)`, a and b bitvector	$\{\texttt{v01}, \texttt{v10}, \texttt{v=-}\}$
$a =$ `(cons c b)`	$\{\texttt{sc>1}, \texttt{s>1}\}$

Table 4.8: Operators that can be captured in a single relation

In addition, vertices in V_T strengthen the relation between the input object and itself from $\{\texttt{tt}, \texttt{ff}\}$ to $\{\texttt{tt}\}$, and vertices in V_F strengthen the relation to $\{\texttt{ff}\}$. So Thinner treats these vertices as single-relation operators too.

The constraint computation

Suppose there is an addition $a =$ `(+ b c)` in the trace graph. The relations among a, b and c can't be captured by directly strengthening entries in the table of relations. But because c is the sum of a and b, not all of the 29^3 triples of primitive integer relations for the pairs $((a, b), (b, c), (a, c))$ are possible. For example, if b and c are both positive, neither can be greater than a; so tuples like $(\{\texttt{++>2+}\}, \{\texttt{++>2+}\}, \{\texttt{++<2+}\})$ can be excluded. Of course, if the semantic analyzer ever developed exactly those relations for a, b and c it would have a contradiction. More significantly, if it ever finds a triple like $(\{\texttt{++>2+}\}, \{\texttt{++>2+}\}, \{\texttt{++<2+}, \texttt{++>2+}\})$ it can strengthen it to $(\{\texttt{++>2+}\}, \{\texttt{++>2+}\}, \{\texttt{++>2+}\})$ just by eliminating the impossible.

Definition 4.1 *Let T be any k-tuple of types. A* constraint (P, S) *consists of a k-tuple P of pairs of objects, in which the two objects in each pair P_i are of type T_i; and a* constraint set *S of k-tuples of expressible relations, with the property that for each $C \in S$, each relation C_i is of type T_i.*

For any k-tuple of types T consider the space of k-tuples of primitive relations on the corresponding types:

$$T\text{-space} = X^{T_1} \times \ldots \times X^{T_k}$$

Any tuple Q of *expressible* relations describes a cube in T-space, for the appropriate tuple of types T:

$$\{(r_1, \ldots, r_k) \mid \bigwedge_{i \in [1..k]} r_i \in Q_i\}$$

A constraint set S is a set of tuples of expressible relations: it describes the region of T-space which is the disjunction of the cubes described by each $C \in S$. The inference procedure takes the current tuple of relations Q for the pairs in P (remember, this describes a cube in T-space) and the constraint set S, and yields the smallest cube R containing the conjunction of Q and S. Each relation in R is at least as strong as the corresponding relation in Q; if it's stronger, we've gained information.

The function Constrain that follows performs the basic constraint computation. Given a current tuple Q of relations and a constraint set S, it returns a potentially strengthened tuple of relations.

function Constrain(Q, S);

Q is a tuple of relations;

S is a set of tuples of relations;

Constrain(Q, S) returns a tuple of relations;

begin

  $R :=$ the tuple of bottom relations;

  for each tuple C in S

    begin

      $I := Q \wedge C$ (the componentwise conjunction);

      if I does not contain a bottom relation **then**

        $R := R \vee I$ (the componentwise disjunction)

    end;

  return R

end;

Constraint sets

Now we have a method for constraining a tuple of relations given a constraint set. The constraint set describes a space of legal tuples of primitives. Where does a constraint set come from? Each constraint set used by Thinner is generated by a program (and tested by another program); almost all the constraint sets are defined before Thinner runs. Most have less than 10 elements, and the largest (the list transitivity constraint set) only 300. (Compare this to the size of the tuple spaces, which contain as many as 29^6 elements).

Thinner uses four types of constraints: functional constraints, transitivity constraints, list property constraints, and transparency constraints.

Functional constraints Some of the operators in TL establish a single binary relation, as described previously; others determine a restriction on the relations among the input and output objects for a vertex. For each such operator in the language there is a fixed constraint set S; for each such vertex in a trace graph we establish a constraint (P, S) that connects the appropriate tuple P of object pairs to the constraint set for that operator.

Integers An addition operation a = (+ b c) and the successor and predecessor relations are intimately related. If we learn that $|c| = 1$, for example, we want to eliminate many of the possible primitive relations between a and b; and if we learn that a is the successor of b, we want to be able to conclude that $c = 1$. To make this kind of inference possible an addition constraint (P, S) doesn't have $P = ((a,b),(b,c),(a,c))$, as you might expect; instead $P = ((a,b),(b,c),(a,c),(a,1),(b,1),(c,1))$.

The constraint set for a subtraction operation b = (- a c) also contains sextuples—in fact, it's the same constraint set as for addition (but with the pairs treated in a different order), since since b = (- a c) if and only if a = (+ b c). The constraint set for a multiplication operation a = (* b c) uses the same trick: sextuples of relations, constraining not only the pairs (a,b), (b,c) and (a,c) but also the pairs $(a,1)$, $(b,1)$ and $(c,1)$. In this case the trick is even more pertinent since 1 is the multiplicative identity.

Booleans The constraint set for the conjunction of booleans is obvious, as shown in Table 4.9. Boolean disjunction is handled similarly.

(b,c)	(c,a)	(b,a)
{ff}	{ff}	{ff}
{ft}	{tf}	{ff}
{tf}	{ff}	{tf}
{tt}	{tt}	{tt}

Table 4.9: The constraint set for boolean conjunction: a = (and b c)

There are comparison operators that have one boolean output and two inputs of other data types: the `equal` operator for every data type, and `<`, `<=`, `>=` and `>` on integers. There are only two tuples in these constraint sets. For example, consider the `<` operator on integers: suppose $a =$ `(<` x y`)`. The tuple of pairs to be constrained is $((x, y), (a, a))$ and there are only two possibilities: $(<, \texttt{tt})$ and $(\geq, \texttt{ff})$. Eliminate one and you have the fullest possible inference; eliminate both and you have a contradiction.

Bitvectors The constraint set for the logical-and of bitvectors is straightforward, as shown in Table 4.10. The logical-or of bitvectors is handled similarly.

(b,c)	(c,a)	(b,a)
`{v00}`	`{v00}`	`{v00}`
`{v01}`	`{v10}`	`{v00}`
`{v10}`	`{v00}`	`{v10}`
`{v11}`	`{v11}`	`{v11}`
`{v0x}`	`{vx0}`	`{v00}`
`{vx0}`	`{v00}`	`{vx0}`
`{v=}`	`{v=}`	`{v=}`
`{v=-,v<-}`	`{vx0}`	`{vx0}`
`{vx1}`	`{v1x}`	`{v=}`
`{v1x}`	`{v=}`	`{v1x}`
`{v>-,vxx}`	`{v>}`	`{v>}`
`{v>}`	`{v=}`	`{v>}`
`{v<}`	`{v>}`	`{v=}`

Table 4.10: The constraint set for bitvector conjunction: $a =$ `(and` b c`)`

Lists Two list operators, `append` and `cdr`, require constraint sets. The constraint set for $c =$ `(append` a b`)` contains sextuples of relations: it constrains (a, b), (b, c) and (a, c) and also the relations of each of a, b and c with the empty list. The constraint set for $b =$ `(cdr` a`)` constrains triples, constraining the relation (a, b)

and also the relations of a and b with the empty list. Thinner could approximate the cdr relation immediately, as {sc<1, s<1, sc=0}, but by constraining relations with the empty list at them same time it gains the power to make further inferences. For example, if a is the empty list Thinner will conclude that b is also the empty list.

Transitivity constraints The relations manipulated by the semantic analyzer have transitive properties. When one relation is strengthened, we want to take advantage of transitivity to strengthen others, even when the objects in question are not neighbors in the graph.

The semantic analyzer treats transitivity as a kind of triangle property: a restriction on the three binary relations among any three objects. There are $O(n^3)$ of these triangles. To keep the complexity down to $O(n^2)$ the relational constraint algorithm installs transitivity constraints on certain "interesting" triples only, leaving most triples unconstrained. A triple (a, b, c) is deemed interesting only if there is a functional constraint on (a, b), or if a is a special constant like 0. My experiments suggest that this captures many of the useful inferences that occur when full transitivity is used.

Even with full transitivity, it may seem at first glance that an important class of inferences having to do with unary relations is missing. For example, suppose we have established that x{0+<2+}y. This fixes a binary relation between x and y, of course, but it also fixes something about each integer independently: x must be 0 and y must be positive. This means that there are restrictions on the relations for (x, z) and (y, z), for any z of that same type.

As it happens, this kind of inference occurs as a by-product of inference about transitivity. Given a relation xRy, a transitivity constraint will apply to the relations in all triangles with (x, y) on one leg, including the triangle whose other two legs are (y, x) and (x, x). So when any relation restricts x's sign, that will be reflected in the reflexive relation for (x, x). (As mentioned above, this starts out as {`--=0`, `00=0`, `++=0`}.) Then, when the relation for (x, x) is strengthened, a transitivity constraint will apply to the relations in all triangles with (x, x) on one leg. The other two legs will be (x, z) and (z, x) for all z of the same type, so the new information about x will be reflected in all relations involving x.

List property constraints Sometimes we know that one object is the car or length of another object. This happens with the empty lists, of course; it also happens when the graph contains `length`, `car` or `cons` operators. Whenever we have two lists a and b that both have length objects or both have car objects, we install a list property constraint. This constraint ensures the relation between a and b, the relation between car(a) and car(b), and the relation between length(a) and length(b), are as strong as possible taken jointly. For example, if we have two list objects a and b and two base-type objects c and d, and if we know that $c = \text{car}(a)$, $d = \text{car}(b)$, and $a = b$, then the list property constraint for a and b excludes the possibility that $c \neq d$.

The relations of a and b with the empty list contain significant information about their cars and lengths. So a list property constraint (P, S) actually applies to 5-tuples of relations:

$$P = ((a, b), (\text{car}(a), \text{car}(b)), (\text{length}(a), \text{length}(b)), (a, \text{nil}), (b, \text{nil}))$$

(I have used "nil" here to refer to an empty list; which empty list depends on the type of a and b.)

Transparency constraints Since the language is purely functional, all functions are referentially transparent: given the same inputs, they produce the same outputs. This applies even to nonterminal functions, about which the semantic analyzer has (or, more accurately, uses) no other information. Given two nonterminal function calls $c_1 = $ `(f` a_1 b_1`)` and $c_2 = $ `(f` a_2 b_2`)`, and given that $a_1 = a_2$ and $b_1 = b_2$, we should be able to conclude that $c_1 = c_2$ without knowing anything about the function f. Or, given the information that $a_1 = a_2$ and $c_1 \neq c_2$, we should be able to conclude that $b_1 \neq b_2$.

To perform this kind of inference, the semantic analyzer places a constraint on the inputs and outputs for each pair of vertices with the same function. A custom constraint set is built to conform to the number and type of the function's inputs and outputs. (This is the only time Thinner constructs a constraint set—all others are prefabricated.) A pair of n-input, m-output vertices with the same nonterminal function will get a constraint on $(n + m)$-tuples.

This method could apply to terminal as well as nonterminal vertices: we could construct a custom "transparency" constraint set for each operator, which in addition to transparency could reflect things like associativity (like `or`), bijectivity (like `cons`), or determination by any two of three values (like `*`). The current version of Thinner doesn't do this.

Optimizing constraint computation

Each constraint (P, S) connects a tuple of pairs of objects to a constraint set. The Constrain algorithm uses the constraint set to strengthen the current tuple of relations for those pairs; and, as we'll see, it is often used to compute the same constraint repeatedly. Since it is being called repeatedly with the same constraint set S and with a tuple Q that gets stronger monotonically, it can be optimized to avoid wasted recomputation. The idea is to have each constraint use its own copy of the constraint set S, initialized as described previously; then when Constrain finds that the intersection of Q with some tuple $C \in S$ is empty, it discards that tuple from S.

```
procedure FastConstrain(Q, S);
    Q is a tuple of relations updated in place;
    S is a set of tuples of relations updated in place;
begin
    R := the tuple of bottom relations;
    T := the empty set;
    for each tuple C in S
        begin
            I := Q ∧ C (the componentwise conjunction);
            if I does not contain a bottom relation then
                begin
                    R := R ∨ I (the componentwise disjunction);
                    T := T ∪ {I}
                end
        end;
    Q := R;
    S := T
end;
```

The initial constraint sets themselves can also be optimized. For example, if a constraint set includes two tuples ({++=0}, {++<1}) and ({++=0}, {++>1}), it can be reduced by removing those tuples and replacing them with ({++=0}, {++<1, ++>1}).

This is simple logic minimization, analogous to the optimization of $(a \wedge b) \vee (a \wedge c)$ to $(a \wedge (b \vee c))$. In fact, when the constraint sets used by Thinner are generated, they are, initially, sets of tuples of primitive relations—it's just easier to generate them that way. But before Thinner sees them they are reduced automatically to minimal sets of cubes.

One might take this further and ask: why not dispense with the general constraint set algorithm and instead implement a logically minimal custom constraint algorithm for each operator? I experimented with this idea by converting a constraint set into logic equations, passing them to a logic minimizer (the Berkeley tool MIS), and converting the resulting logic equations directly into code by hand. The result, unfortunately, was a somewhat slower constraint procedure! Although it was logically minimal it lacked the regular tight-looping structure of the general procedure.

4.2.3 The relational constraint algorithm

The difficult part of the relational constraint method is the design of suitable vocabularies of relations and the construction of appropriate constraint sets. The relational constraint algorithm itself is quite simple, and the reader has no doubt anticipated it. This section presents the algorithm, gives correctness and complexity results, and offers some tips on implementation.

The algorithm

Here's the algorithm in a nutshell. First, establish a table of relations indexed by pairs of objects and initialize each relation in the table. Next, install the constraints: functional constraints, transitivity constraints, transparency constraints, and list property constraints, as described above. Initialize a set of constraints to be computed; initially, this set contains all the constraints. Now loop while there are more constraints to be computed: choose one, compute it, and update the table of relations; for any entry that got stronger, find the constraints that apply to that pair and add them to the set of those to be computed; and remove the one we just performed.

We need to be able to find all the constraints that apply to a given pair in constant time, so we'll construct a table *Con* indexed by pairs of objects, giving a set of relevant constraints for each pair. The AddCon procedure installs a constraint in the right sets, and also adds the constraint to the set of constraints to be computed.

procedure AddCon((P, S), *Con*, *Todo*);

  (P, S) is a constraint;

  Con is a table of sets of constraints, *updated in place*;

  Todo is a set of constraints, *updated in place*;

begin

  for each pair (i, j) in P

    $Con(i, j) := Con(i,j) \cup \{(P, S)\}$;

  $Todo := Todo \cup \{(P, S)\}$

end;

Whenever we strengthen a relation in the table of relations, we also strengthen the reversed relation. (We'll assume that the new relation is stronger than the old relation.) Then we find all the constraints that apply to either and include them in the set of constraints to be computed.

procedure Strengthen(i, j, r, *Con*, *Rel*, *Todo*);

i and j are objects;

r is a relation;

Con is a table of sets of constraints;

Rel is a table of relations, *updated in place*;

Todo is a set of constraints, *updated in place*;

begin

$Rel(i, j) := r$;

$Rel(j, i) := r$ reversed;

for each constraint (P', S') in $Con(i, j)$ or $Con(j, i)$

$Todo := Todo \cup \{(P', S')\}$

end;

The installation of all the constraints has one subtlety, already noted: we don't install all the possible transitivity constraints, only those that involve pairs that also have functional constraints. The Init procedure takes a graph G and a set of objects *Obs* and initializes the *Con* table, the table of relations *Rel*, and the set *Todo* of constraints to be computed.

procedure Init(G, *Obs*, *Con*, *Rel*, *Todo*);

G is a trace graph;

Obs is the collection of objects for G;

Con is a table of sets of constraints, *updated in place*;

Rel is a table of relations, *updated in place*;

Todo is a set of constraints, *updated in place*;

begin

$Todo :=$ the empty set;

```
for each pair of identically typed objects (i, j) from Obs
    begin
        Con(i, j) := the empty set;
        Rel(i, j) := the initial relation for (i, j) (Section 4.2.2)
    end;
for each single-relation vertex v in G
    update the appropriate Rel(i, j);
for each vertex v in G with a functional constraint
    AddCon({the functional constraint for v}, Con, Todo);
for each pair of identically typed objects (i, j) from Obs
    if i is a special constant or Con(i, j) is not empty then
        for each object k from Obs of the same type as i and j
            AddCon({the transitivity constraint on (i, j, k)},Con,Todo);
for each pair of identically typed list objects (i, j) with car or length objects
    AddCon({the list property constraint for (i, j)},Con,Todo);
for each pair of vertices (v, w) in G with identical nonterminal functions
    AddCon({the transparency constraint for (v, w)},Con,Todo)
end;
```

The input to the main procedure RelationalConstraint is a trace graph G; we'll assume that we are also given *Obs*, the set of objects of interest.

```
procedure RelationalConstraint(G, Obs);
    G is a trace graph;
    Obs is the collection of objects for G;
    Con is a table of sets of constraints;
    Rel is a table of relations;
```

```
    Todo is a set of constraints;
begin
1   Init(G, Obs, Con, Rel, Todo);
2   while more constraints in Todo do
3       begin
4           (P, S) := a constraint from Todo;
5           Q := the tuple giving the relation Rel(i, j) for each (i, j) in P;
6           Q := Constrain(Q, S);
7           for each pair (i, j) in P and corresponding relation r in Q
8               if r is stronger than Rel(i, j) (that is, if Rel(i, j) ⇏ r) then
9                   Strengthen(i,j,r,Con,Rel,Todo);
10          remove (P, S) from Todo
11      end
end;
```

Correctness

Theorem 4.1 *The RelationalConstraint algorithm terminates with a table of relations Rel that cannot be strengthened by the given constraints.*

Proof: An invariant for the loop at line 2 in RelationalConstraint is this: if the computation of a constraint would strengthen any relation in *Rel*, that constraint is in *Todo*. This invariant is established by Init, which places all constraints in *Todo*. The invariant is reestablished inside the loop, since whenever any entry in *Rel* is altered, all constraints involving that pair are added to *Todo*; and the one entry removed from *Todo* is the one that was just computed (which would not result in any further strengthening if recomputed).

Let c be the number of constraints installed by Init, and let $k(Rel)$ be the sum over all pairs (i, j) of the distance in the lattice of relations from $Rel(i, j)$ to the bottom relation for that type. Then consider $t = (c+1) \cdot k(Rel) + |Todo|$. On any pass through the loop either some relation is strengthened or no relation is strengthened. If some relation is strengthened then t decreases since $k(Rel)$ is reduced (even though $|Todo|$ may grow as large as c). If no relation is strengthened then t decreases since one constraint is removed from *Todo* and none are added. So t always decreases; and when $t = 0$ (if not before) we have $|Todo| = 0$ and the loop at line 2 terminates. Since *Todo* is empty at this point, we can conclude from the invariant that no constraint would strengthen any relation in *Rel* any further. $\square$

One might wonder whether the final table found by RelationalConstraint is unique for a given initial table and collection of constraints. Since we haven't specified the order in which constraints are selected for computation, is it possible that different orders yield different solutions? It turns out that the order in which constraints are computed is irrelevant (although it makes a difference in efficiency, as noted in the Chapter 6), and RelationalConstraint always finds the same solution.

To prove this, note that an n-by-n table of expressible relations, like the Rel table computed in RelationalConstraint, defines a cube in the space of n^2-tuples of primitive relations. Each individual constraint can be promoted to this super tuple-space by reorganizing the tuples and adding $\top$ elements for those dimensions not previously represented. So we can think of the method of constraints as starting with some cube A (the cube defined by the initial Rel table) and restricting it in a series of steps to reach some final cube B. We'll use $\subseteq$ to denote the subcube relation—a cube A is a subcube of a cube B if and only if each relation in the table for A is equal to or stronger than the corresponding relation in the table for B. Each step in the main loop of RelationalConstraint is a constraint computation, which is, as we've seen, the computation of the smallest cube that contains the intersection of the current cube with the region defined by a constraint set. If S is a sequence of constraints that takes initial cube A to final cube B, we'll write $A \rightarrow_S B$.

Lemma 4.2 *If $A \rightarrow_S B$ then $B \subseteq A$.*

Lemma 4.3 *If $A \rightarrow_S B$ and $A' \rightarrow_S B'$, and if $A \subseteq A'$, then $B \subseteq B'$.*

Proof: both lemmas follow immediately from the observation that each step in S is a constraint that forms the smallest cube containing the intersection of the previous cube with a region of the tuple space. □

Theorem 4.4 *For a given initial cube and collection of constraints, Relational-Constraint finds a unique final cube.*

Proof: Suppose we have two runs of RelationalConstraint with the same collection of constraints and the same initial cube A. Let S and R be the two sequences of constraints from these runs, so $A \rightarrow_S B$ and $A \rightarrow_R B'$. Apply the sequence R to B. By Theorem 4.1, none of the constraints will have any effect, so $B \rightarrow_R B$. By Lemma 4.2 we know $B \subseteq A$, and since $A \rightarrow_R B'$ we can conclude from Lemma 4.3 that $B \subseteq B'$. Now apply the sequence S to B'. By Theorem 4.1, none of the constraints will have any effect, so $B' \rightarrow_S B'$. Reasoning as before we can conclude $B \subseteq B'$. Since $B \subseteq B'$ and $B' \subseteq B$, we have $B = B'$: RelationalConstraint finds a unique final cube. □

This is as much as we can say about correctness: that the algorithm terminates with a unique table of relations that cannot be further strengthened by the constraints. A more thorough proof of correctness would depend on the correctness and completeness of the constraint sets.

Complexity analysis

Let n be the number of vertices in the graph. We assume that the trace graph has bounded degree: that is, that functions have bounded input and output arity and that the fanout is bounded. So the number of vertices in the graph, the number of edges in the graph, and the number of objects for the graph, are within constant factors of each other.

Theorem 4.5 *RelationalConstraint runs in $O(n^2)$ time.*

Proof: First we need to determine how many constraints are installed by Init. The number of functional constraints installed at lines 9-10 is n—one for each vertex. At lines 11-14 $O(n^2)$ transitivity constraints are installed: $O(n)$ constraints for each of the $O(n)$ pairs constrained in the previous step. At lines 15-16 $O(n^2)$ list property constraints are installed, at most one for each pair of list objects. Finally, at lines 17-18 $O(n^2)$ transparency constraints are installed (normally far fewer than n^2 since the number of pairs of vertices with identical nonterminal functions is likely to be small). The total number of constraints installed by Init is $O(n^2)$, and the time in initialization (line 1 in RelationalConstraint) is also $O(n^2)$.

We now turn to the main loop in RelationalConstraint. The number of times the test at line 8 can be satisfied for a given pair (i, j) is bounded by the height of the lattice of relations, which is constant. So Strengthen is called O(1) times for each pair (i, j). The total time spent in Strengthen is therefore $O(\sum_{(i,j)} |Con(i, j)|) =$

$O(n^2)$. A constraint is computed at most once initially, and then at most once every time one of its pairs is strengthened; this can happen for a given constraint only a constant number of times, so the loop in lines 2-11 is executed $O(n^2)$ times. Disregarding Strengthen, whose contribution has already been counted, each pass through the loop takes constant time. So the total time in the main loop—and the total time in RelationalConstraint—is $O(n^2)$. □

Full computation of transitivity, which is an option left turned off in Thinner, can be achieved by skipping the test at line 12 in Init. With full transitivity the complexity of the algorithm rises to $O(n^3)$.

4.2.4 Examples

In this section we will look at two simple examples of relational constraint. The second example is actually not so simple; it demonstrates that in spite of its low complexity, relational constraint sometimes proves surprisingly tricky things.

A simple integer example

Consider the following program:

```
(defun f (a b)
    (>= (+ (* a a) b) b))
```

Figure 4.1 shows the corresponding trace graph. There are six integer objects for inference: the input variables a and b, the intermediate values labeled x and y, and the constants 0 and 1. There are also three boolean objects: the output value labeled z and the constants true and false.

The initial table of relations is shown in Table 4.11. Here r_0 is the initial relation between the constant 0 and any nonconstant integer object and r_1 is the

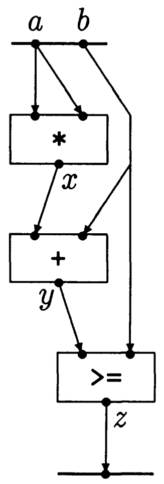


Figure 4.1: Trace graph for the simple integer example

initial relation between the constant 1 and any nonconstant. Note that the initial array is symmetric, with relations at least as strong as equality on the diagonal. After relational constraint, the final table of relations is as shown in Table 4.12; empty slots indicate unchanged relations.

We've concluded that the output of this function is a constant, the boolean value true. How did this happen? There's no point in reconstructing the actual order of events (since we haven't specified the order in which waiting constraints are performed), but we can analyze the results logically.

The functional constraint for multiplication constrains the relations on pairs (a, a), (a, x), $(a, 1)$ and $(x, 1)$ to make them consistent with $x = a \cdot a$; in particular, this constraint tightens the relation on (a, x) from $\top$ to r_4. Note that this subsumes the "law of signs" in that any x satisfying ar_4x must be non-negative. That's not all ar_4x tells us: the magnitude of $x = a \cdot a$ must be either equal to the magnitude of a (this happens when $|a| \leq 1$) or greater than the magnitude of a by at least 2 (this happens when $|a| > 1$). But for our purposes the important thing is that x is non-negative, and after some transitivity constraints this fact is reflected throughout

	0	1	a	b	x	y	true	false	z
0	00=0	0+<1	r_0	r_0	r_0	r_0			
1	+0>1	++=0	r_1	r_1	r_1	r_1			
a	rev(r_0)	rev(r_1)	$=$	$\top$	$\top$	$\top$			
b	rev(r_0)	rev(r_1)	$\top$	$=$	$\top$	$\top$			
x	rev(r_0)	rev(r_1)	$\top$	$\top$	$=$	$\top$			
y	rev(r_0)	rev(r_1)	$\top$	$\top$	$\top$	$=$			
true							tt	tf	{tf,tt}
false							ft	ff	{ff,ft}
z							{ft,tt}	{tf,ff}	$=$

$$
\begin{aligned}
r_0 &= \{\texttt{0-<1}, \texttt{0-<2+}, \texttt{00=0}, \texttt{0+<1}, \texttt{0+<2+}\} \\
r_1 &= \{\texttt{+-<1}, \texttt{+-<2+}, \texttt{+-=0}, \texttt{+0>1}, \texttt{++=0}, \texttt{++<1}, \texttt{++<2+}\}
\end{aligned}
$$

Table 4.11: Initial relations for the simple integer example

	0	1	a	b	x	y	true	false	z
0					r_2				
1					r_3				
a					r_4				
b					r_5	$\leq$			
x	rev(r_2)	rev(r_3)	rev(r_4)	rev(r_5)	r_6	rev(r_5)			
y				$\geq$	r_5				
true									tt
false									ft
z							tt	tf	tt

$$
\begin{aligned}
r_2 &= r_0 \setminus \{\texttt{0-<1}, \texttt{0-<2+}\} \\
r_3 &= r_1 \setminus \{\texttt{+-<1}, \texttt{+-<2+}, \texttt{+-=0}, \texttt{++<1}\} \\
r_4 &= \{\texttt{-+<2+}, \texttt{-+=0}, \texttt{++<2+}, \texttt{00=0}, \texttt{++=0}\} \\
r_5 &= r_4 \cup \{\texttt{-0>1}, \texttt{-0>2+}, \texttt{-+>1}, \texttt{-+>2+}, \texttt{+0>2+}, \texttt{+0>1}, \\
&\qquad \texttt{++>2+}, \texttt{++>1}, \texttt{-+<1}, \texttt{0+<2+}, \texttt{0+<1}, \texttt{++<1}\} \\
r_6 &= \{\texttt{00=0}, \texttt{++=0}\}
\end{aligned}
$$

Table 4.12: Final relations for the simple integer example

the table, affecting x's entire row and column. This explains r_2 and r_5: they are the original relations strengthened for $x \geq 0$. (r_3 is even stronger: note that it does not allow x, the square of an integer, to be 2.)

Now the functional constraint for addition constrains the relations on pairs (x, b), (x, y), (b, y), $(x, 1)$, $(b, 1)$ and $(y, 1)$ to make them consistent with $y = x + b$. Since x is positive, this constraint tightens the relation on (b, y) from $\top$ to $\leq$. This in turn triggers the computation of the functional constraint for the `<=` test, which constrains the relations on (y, b) and (z, z): since $y \geq b$, $z =$ true. Finally, transitivity strengthens the relations between z and the constants true and false.

A tricky list example

Consider the following program:

```
(defun g (a b)
  (let ((c (cons 1 a))
        (d (cons 0 inil)))
    (= (append c d) (append b c))))
```

Figure 4.2 shows the corresponding trace graph. Thinner decides that the output of this program is always the boolean value false. The reader is invited to prove this, using any method, before proceeding.

Perhaps the reader found this inductive proof-by-contradiction. List x is $1 \cdot a \cdot 0$ and list y is $b \cdot 1 \cdot a$. Suppose $x = y$. Then $|b| = 1$, a cannot be inil and its last element must be 0. We also observe that if $a_i = 0$ then $a_{i-1} = 0$ and so, by induction, all the elements of a must be 0. But if $x = y$ the first element of a must be 1. By contradiction, $x \neq y$.

How does Thinner prove this without induction or contradiction? It reasons about seven integer-list objects: the inputs a and b, the "let" variables c and d,

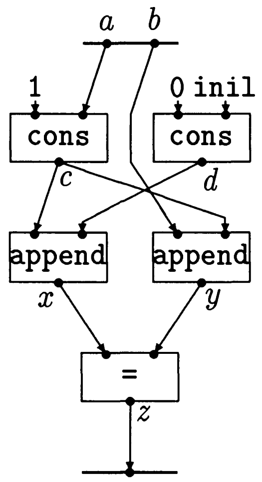


Figure 4.2: Trace graph for the tricky list example

the intermediate values x and y, and the empty integer list. There are also three boolean objects: the output z and the boolean constants true and false. The initial table of relations is shown in Table 4.13. Here r_0 is the initial relation between the empty list and any list, and r_1 is the relation established by `cons` between its input and output lists. Table 4.14 shows the final table of relations.

The `cons` vertices establish initially that 1 is that car of c and 0 is the car of d, and that the length of d is the successor of the length of inil (that is, the length of d is 1). Then because the integer relation 0{0+<1}1 is present (though not shown in the table) the list property constraint for c and d yields r_4, which says (among other things) that c and d have different cars.

The functional constraint for the `append` that computes x constrains the relations on (c, d), (c, x), (d, x), (c, inil), (d, inil) and (x, inil). The constraint set for `append` reflects the fact that if the length of d is 1 and if c and d have different cars, c cannot be a suffix of (`append` c d). (This is where that tricky proof is hiding— it's implicit in the constraint set for `append`.) So when this constraint is computed, we get the relation c<1 between c and x shown in the table. It says

	inil	a	b	c	d	x	y	true	false	z
inil	$=$	r_0	r_0	r_0	r_1	r_0	r_0			
a	rev(r_0)	$=$	$\top$	r_1	$\top$	$\top$	$\top$			
b	rev(r_0)	$\top$	$=$	$\top$	$\top$	$\top$	$\top$			
c	rev(r_0)	rev(r_1)	$\top$	$=$	$\top$	$\top$	$\top$			
d	rev(r_1)	$\top$	$\top$	$\top$	$=$	$\top$	$\top$			
x	rev(r_0)	$\top$	$\top$	$\top$	$\top$	$=$	$\top$			
y	rev(r_0)	$\top$	$\top$	$\top$	$\top$	$\top$	$=$			
true								`tt`	`tf`	{`tf`,`tt`}
false								`ft`	`ff`	{`ff`,`ft`}
z								{`ft`,`tt`}	{`tf`,`ff`}	$=$

$$
\begin{aligned}
r_0 &= \{\texttt{sc=0}, \texttt{sc<2+}, \texttt{sc<1}, \texttt{s<2+}, \texttt{s<1}\} \\
r_1 &= \{\texttt{sc<1}, \texttt{s<1}\}
\end{aligned}
$$

Table 4.13: Initial relations for the tricky list example

that c is one shorter that x, that they have the same car, but that c is not a suffix of x.

On the other hand, the append that computes y constrains its tuple of relations to assert that c is a suffix of y (relation r_0). Since c is a suffix of y and not a suffix of x, we conclude by transitivity that x and y cannot be equal; and the functional constraint on the equality test settles the value of z.

4.2.5 Comparison with other methods

Methods of greater power

One way to reason about programs is to use a more powerful inference system: perhaps a Prolog-style backward-chaining system, or something that finds simple inductive proofs like Boyer and Moore's early system for proving theorems about Lisp programs [BM75]; maybe even a powerful reasoning system like Nuprl [C+86].

My early experiments with a prototype of Thinner used a simple backward-

	inil	a	b	c	d	x	y
inil				r_2	sc<1	s<2+	r_3
a				r_1	r_8	r_5	r_6
b					r_8		r_7
c	rev(r_2)	rev(r_1)			r_4	c<1	r_0
d	sc>1	rev(r_8)	rev(r_8)	rev(r_4)		r_2	r_9
x	s>2+	rev(r_5)		c>1	rev(r_2)		r_{10}
y	rev(r_3)	rev(r_6)	rev(r_7)	rev(r_0)	rev(r_9)	rev(r_{10})	

	true	false	z
true			tf
false			ff
z	ft	ff	ff

$$
\begin{aligned}
r_2 &= r_0 - \{\texttt{sc=0}, \texttt{sc<2+}, \texttt{sc<1}\} \\
r_3 &= r_0 - \{\texttt{sc=0}, \texttt{sc<1}\} \\
r_4 &= \{\texttt{=0}, \texttt{s>2+}, \texttt{s>1}, \texttt{>2+}, \texttt{>1}\} \\
r_5 &= \{\texttt{sc<2+}, \texttt{s<2+}, \texttt{c<2+}, \texttt{<2+}\} \\
r_6 &= r_0 - \{\texttt{sc=0}\} \\
r_7 &= r_6 \cup \{\texttt{c<2+}, \texttt{c<1}\} \\
r_8 &= r_4 \cup \{\texttt{sc=0}, \texttt{sc<1}, \texttt{sc>2+}, \texttt{sc>1}, \texttt{c>2+}, \texttt{c>1}\} \\
r_9 &= r_5 \cup \{\texttt{=0}, \texttt{c<1}, \texttt{<1}, \texttt{s<1}\} \\
r_{10} &= \{\texttt{c>1}, \texttt{c=0}, \texttt{=0}, \texttt{c<2+}, \texttt{c<1}, \texttt{<2+}, \texttt{<1}\}
\end{aligned}
$$

Table 4.14: Final relations for the tricky list example

chaining inference engine. My choice of the vocabulary of relations used by the current Thinner originated in post-mortem examinations of the most useful proofs discovered by that prototype. Looking over these proofs, I made several other observations:

- A trace graph suggests natural paths of inference that a general purpose inference engine ignores. For example, if you've just concluded that $a = 0$, the next natural step is to reason about uses of a in the graph.

- Thinner needs to examine programs and discover opportunities for thinning—and this means forward, not backward, chaining. What's more, Thinner doesn't need proofs, it only needs conclusions.

- The number of objects in a trace graph is fixed and rather small, and it is these objects about which Thinner needs to reason. There is no real need for unification.

These observations led me away from general-purpose inference techniques which, although powerful, are too costly (that is, slow) for the inference task Thinner faces.

Methods of greater specificity

The literature contains quite a few techniques for dealing with the integer variables of imperative programs. Constant propagation is a time-honored technique [WZ91]; it treats program variables independently, without considering the relations between them. Cousot and Cousot extended this kind of analysis to constant intervals [CC76].

Relations between integer variables have also been studied: the method of Karr [Kar76] develops a system of affine equations on a program's variables, and the linear restraint analysis of Cousot and Halbwachs [CH78] develops a system of affine inequalities. A strong motivation in this area has been the compiler-writer's desire to vectorize loops automatically, which requires congruence analysis. In this vein are Granger's congruence analysis, which yields assertions of the form $x \equiv c[m]$ [Gra89] and, later, analysis of congruence equalities, which yields a system of congruence equations [Gra91].

These techniques are too specialized for use in Thinner, which needs to reason about other data types as well. Many are also too expensive: congruence analysis is $O(pn^6 \log^2 n)$ where p is the program size and n the number of integer variables, while linear restraint analysis is exponential.

4.2.6 Abstract interpretation

Many of the inference techniques discussed above appear in the literature as *abstract interpretations.* Abstract interpretation is a framework for the formal treatment of program analysis techniques, first advanced by Cousot and Cousot [CC77, CC79]; see Abramsky and Hankin [AH87] for a modern introduction. Techniques for the analysis of functional programs that have been expressed in this framework include strictness analysis for the optimization of lazy languages [Hug87,Wad87] and update-in-place analysis for the optimization of storage reuse [Hud87]. Not wanting to appear behind the times, I have sketched out a treatment of relational constraint in this same same style. In fact, I have sketched a treatment in two styles: the original Cousot and Cousot treatment using adjoined functions, and a more recent style involving semantic schemata, due to Mycroft.

Abstract interpretation with adjoined functions

First, the original Cousot and Cousot treatment using adjoined functions. My terminology and notation follow theirs [CC79].

Let $E(G)$ denote the set of executions of graph G, with respect to any mapping $\mathcal{F}$ from nonterminals to functions. An *assertion* about a given trace graph G is a total function mapping executions of G to $\{\text{true}, \text{false}\}$. Let $\mathcal{A}_G$ be the set of assertions about graph G:

$$\mathcal{A}_G : E(G) \rightarrow \{\text{true}, \text{false}\}$$

$\mathcal{A}_G$ ordered by implication forms a complete boolean lattice.

Recall that in Chapter 3 we discussed a function Lang that maps any trace graph G and trace grammar Q into the set of all executed trace graphs derivable from G in Q. That is, $\text{Lang}(G, Q)$ is a set of pairs (G', f') where f' is an execution of G' and $G \Rightarrow_Q^* G'$. We can use this same function to identify, for each graph G and grammar Q, the particular assertion giving a standard semantics: the assertion $\mathcal{C}_{G,Q}$, which is true of exactly those executions f of G that match the execution f' for some $(G', f') \in \text{Lang}(G, Q)$. That is: if $m_{G \rightarrow G'}$ is the function mapping each object of G to the corresponding object of G',

$$\mathcal{C}_{G,Q} = \lambda f \,.\, [\exists (G', f') \in \text{Lang}(G, Q) \,.\, f = f' \circ m_{G \rightarrow G'}]$$

This is a standard interpretation of trace graphs and grammars.

Now for the abstract interpretation: let A_G be the set of tables of expressible relations on pairs of objects in G, as defined in Section 4.2.1. This is a complete lattice under the partial order $R \sqsubseteq S \equiv \forall (i, j). R(i, j) \sqsubseteq S(i, j)$, using component-wise conjunction and disjunction, the all-$\top$ table as the top element, and the all-$\bot$ table as the bottom element. Elements of A_G represent approximate assertions.

The function $\gamma_G : A_G \to \mathcal{A}_G$, defined as follows, gives meaning to each element of A_G.

$$\gamma_G(R) = \lambda f.[\forall (i,j) \,.\, f(i) R_{(i,j)} f(j)]$$

It should be clear that γ_G is not one-to-one: any table R with a bottom element specifies the empty assertion, and any table R that can be strengthened by transitivity specifies the same assertion before and after strengthening. (This is where the transitivity and symmetry computations of relational constraint enter the picture: they find the pairwise-strongest $R' \in A_G$ with $\gamma_G(R') = \gamma_G(R)$.) The range of γ_G is a set of assertions $\overline{A}_G \subseteq \mathcal{A}_G$: these are the approximate assertions.

Theorem 4.6 *For all assertions $p \in \mathcal{A}_G$ the set $\{q \in \overline{A}_G \mid p \Rightarrow q\}$ has a least element.*

Proof: $\overline{A}_G$ contains the supremum of $\mathcal{A}_G$. It is also closed under conjunction: the assertion given by the table R and the assertion given by the table S are both satisfied if and only if the assertion given by $R \wedge S$ is satisfied—remember that for any two expressible relations their conjunction is also an expressible relation. It follows that $\overline{A}_G$ is a Moore family and any $p \in \mathcal{A}_G$ has a least upper approximation in $\overline{A}_G$. □

We can therefore define an approximation operator $\rho_G : \mathcal{A}_G \to \overline{A}_G$ as

$$\rho_G(p) = \vee\{q \in \overline{A}_G \mid p \Rightarrow q\}$$

To get a representation for the least upper approximation of an assertion p we need to pick one of the representations R such that $\gamma_G(R) = \rho_G(p)$. We can just choose the conjunction over all such R, and so define a function $\alpha_G : \mathcal{A}_G \to A_G$ as:

$$\alpha_G(a) = \wedge\{R \mid \gamma_G(R) = \rho_G(p)\}$$

Now $\langle \alpha_G, \gamma_G \rangle$ is a pair of adjoined functions in the sense of Cousot and Cousot.

Cousot and Cousot translated the operators of their language into predicate transformers. In contrast, we have treated TL operators as predicate constraints: each operator in a graph G yields a constraint (P, S) giving a tuple of pairs P and a description of the space of tuples of relations S that must be satisfied by any table of relations $R \in \overline{A}_G$ consistent with the standard semantics—that is, any R for which $\mathcal{C}_{G,Q} \Rightarrow \gamma_G(R)$.

Abstract interpretation with schematic semantics

A different approach to abstract interpretation is advanced by Mycroft [Myc87]. In this approach, a schematic denotational semantics is presented for the language. One way of completing the semantics yields a standard interpretation, while another way yields an abstract interpretation.

In Chapter 3 we developed a schematic denotational semantics for TL. Recall that we constructed a standard semantics from this schematic semantics by defining the semantic function $\mathcal{T} : \text{Ter} \to \mathcal{C}$ (so that the meaning of each terminal function identifier was the relation normally associated with the corresponding function on values), and by defining the auxiliary function Lang : $Z \to \mathcal{P}_f(N) \to \mathcal{P}(X)$ (so that it generated the set of fully terminal trace graphs from a trace grammar in the usual way). We will now construct an abstract semantics by defining these two functions in a more relaxed way. To distinguish the standard semantic functions from the abstract ones, we will subscript the former with an S and the latter with an A.

First, a definition for Lang_A: let $\text{Lang}_A(G, Q)$ be $\{(G, f) \mid f \text{ is an execution of } G\}$, again allowing executions with respect to any mapping $\mathcal{F}$ from nonterminals to functions. Note that this ignores the grammar Q: it specifies an immediate semantics for trace graph G, whether or not G is fully terminal.

Next, a definition for $\mathcal{T}_A$: to make this definition we will use two auxiliary semantic functions $\mathcal{X}_A$ and $\mathcal{Y}_A$. The first maps a terminal function and a tuple of values to a tuple of pairs of values, and the second maps a terminal function to a set of tuples of primitive binary relations (recall that X^t is the set of primitive relations for each type t).

$$\mathcal{X}_A : \mathrm{T} \to V^* \to (V \times V)^*$$
$$\mathcal{Y}_A : \mathrm{T} \to \mathcal{P}((X^{\mathrm{int}} + X^{\mathrm{bool}} + \ldots)^*)$$

These functions are derived from the constraints (P, S) described previously. $\mathcal{X}_A[\![\mathrm{T}]\!]$ is the function that organizes a tuple of values (giving the inputs and outputs of the primitive function T) into pairs corresponding to the pairs of objects P in a constraint (P, S) for function T. $\mathcal{Y}_A[\![\mathrm{T}]\!]$ is the space of tuples of primitive relations described by the constraint set S in a constraint (P, S) for function T.

Now the semantic function $\mathcal{T}_A$ can be defined as the set of all tuples of values that satisfy the constraint defined by $\mathcal{X}_A$ and $\mathcal{Y}_A$:

$$\mathcal{T}_A[\![\mathrm{T}]\!] = \{\boldsymbol{v} \in V^* | \bigvee_{r \in \mathcal{Y}_A[\![\mathrm{T}]\!]} \bigwedge_i (\mathcal{X}_A[\![\mathrm{T}]\!]\boldsymbol{v})_{i,0}\, r_i\, (\mathcal{X}_A[\![\mathrm{T}]\!]\boldsymbol{v})_{i,1}\}$$

If the constraints described by $\mathcal{X}$ and $\mathcal{Y}$ are constructed correctly we will have $\mathcal{T}_S[\![\mathrm{T}]\!] \subseteq \mathcal{T}_A[\![\mathrm{T}]\!]$ for all terminal functions T.

Because $\mathcal{T}_S[\![\mathrm{T}]\!] \subseteq \mathcal{T}_A[\![\mathrm{T}]\!]$ for all terminal functions T, and because nonterminal functions are unconstrained, it follows that we have an abstract semantics in the sense that if $(G, f) \in \mathcal{Q}_S[\![\mathrm{P}]\!]\mathrm{E}$ then there is a corresponding $(G', f') \in \mathcal{Q}_A[\![\mathrm{P}]\!]\mathrm{E}$.

4.2.7 Discussion

The relational constraint method should be easy to parallelize. Since the order in which constraints are computed is largely irrelevant, there is no reason why they cannot be computed in parallel. Updates to the table of relations are also unusually order-independent since relations grow stronger monotonically.

Relational constraint need not be the only method of inference used in a given system. Thinner's semantic analyzer also uses the normal forms of expressions to gain information about a program (although this was disabled for the examples discussed above). Even a more powerful inference system—say, one with a method for induction—might well seed its initial collection of assertions with those discovered efficiently by the method of relational constraint.

4.3 Normal forms

Thinner's semantic analyzer gains additional information by computing and comparing normal forms of expressions. Whenever the normal expressions for two objects are identical, the objects must be equal. As this is a well-known technique I will not labor it here. For completeness, however, this section outlines the normal forms used by Thinner for each data type.

Inputs to a trace graph and outputs from nonterminal functions within it are always among the variables that may occur in normal expressions, except that if two such objects are known to be equal at the time the normal expressions are computed, one variable will be used for both. Additional variables are created for any object whose normal expression could not be computed in terms of existing variables; these new variables may occur in the normal expressions of the descendents of that object in the graph.

The semantic analyzer enters the normal expression for each object into a hash table. Whenever it discovers two identical normal expressions it asserts the relation of equality between the two corresponding objects. This assertion is just a strengthening of an entry in the table of relations built by RelationalConstraint. (It may trigger additional constraint computations.) Information other than equality might also be extracted from the normal expressions; for example, a variety of relations between two bitvectors could be gleaned from an inspection of their normal expressions. But the semantic analyzer does not attempt this.

4.3.1 Integers

The normal form for integer expressions is a polynomial, represented in recursive form. The variables of these polynomials are the integer objects for which no normal expression is known: the integer inputs to a trace graph, the integer outputs of `abs` and `length` vertices, and the integer outputs of nonterminal vertices. The semantic analyzer computes normal expressions for the integer results of addition, subtraction, multiplication and negation.

4.3.2 Bitvectors

The normal form for bitvectors is a sorted list of variables x_1 through x_n, followed by a list of 2^n bitvector constants. Each constant determines one disjunct in an expression for the object. Let i be an integer from 0 to $2^n - 1$ and suppose the n bits $b_1 \ldots b_n$ give i's binary representation. Then the i'th constant c_i in the normal form of an object contributes a term of the form

$$(c_i \wedge l_1 \wedge \cdots \wedge l_n)$$

where $l_j = x_j$ if $b_j = 0$, and $l_j = \overline{x_j}$ if $b_j = 1$. The disjunction of all 2^n such terms is an expression for the object. Normal expressions are simplified to eliminate unnecessary variables: that is, no variable in a normal expression has the property that the coefficient of each term is the same as the coefficient of that term with the variable complemented. If there are no variables in the list, there is $2^0 = 1$ coefficient: this is the representation for bitvector constants. (The mathematical properties of this form are discussed by Lausch and Nöbauer [LN73]. They prove that every boolean function has exactly one representation in this form.)

The size of a normal expression is exponential in the number of variables, so the semantic analyzer computes only the shorter ones. It places a limit on the number of variables (currently this limit is 7, which limits the size of a normal expression to about 512 bytes). It proceeds in a breadth-first fashion: when it has canonical forms for the inputs to a bitvector primitive, it determines the total number of distinct variables in the inputs. If that number is greater than the limit, it establishes a new variable for the output of that primitive. Otherwise, it computes a normal expression for the output of the primitive using the normal expressions of the inputs, then simplifies the result to eliminate any unnecessary variables.

Given an object a with a normal expression, it is easy to compute a normal expression for the complement of a: just complement each constant. Given two objects a and b with normal expressions over the same variables in the same order, it is easy to compute a normal expression for the and or or of a and b: simply logical-and or logical-or the corresponding constants together. When a and b have normal expressions over different sets of variables, it is only slightly more complicated to choose the right pair of constants to logical-and or logical-or for each constant in the result.

4.3.3 Booleans

The normal form for booleans is the same as the normal form for bitvectors, except that the 2^n constants for each expression are boolean constants and not bitvector constants. Since this can reduce the space consumed by normal forms by a factor of 2^5, it is possible to allow more variables.

Many comparison primitives—for example, integer equality—take nonboolean inputs and produce a boolean output. That boolean output is treated as a variable in the normal expressions of its descendants.

4.3.4 Lists

In considering the normal form for lists, it is important to remember that TL lists are not the same as Lisp lists. TL only allows the three flat list types: `(listof integer)`, `(listof boolean)`, and `(listof bitvector)`.

The list normal form specifies a list type and a list of terms. Each term is either a variable, a base-type constant, or a car or cdr operation applied to another list normal form. A list of these terms represents the catenation of the elements contributed by each one; this catenation may have occurred in the program as a `cons` or `append`, but the distinction is irrelevant since lists are strongly typed in a way that does not allow lists of lists. Each term is simply a contributor of zero or more list elements. Thus, `(append (cons b inil) n)` and `(cons b n)` have the same normal form: the type `(listof integer)` followed by a list of terms giving a term for `b` followed by a term for `n`.

Any base-type constant or base-type variable is a "solid", known to contribute exactly one element to the expression in which it occurs. In TL the car of a list is always an atom of the base type for that list: the car of an empty list of integers, for example, is zero. So car expressions are also solid. In contrast, a cdr expression or a variable of list type may contribute any number of elements including 0. Knowing whether a term is solid lets us simplify list expressions in several ways. When the first term in a form is solid, the car of that expression is simply that first term and the cdr of that expression is the remainder of the list of terms. For example, `(car (cons (car l) m))` and `(car l)` have the same normal forms, as do `(cdr (append (cons a inil) n))` and `n`.

If the first term in an expression is not solid, we may still be able to simplify when taking the car of that expression: in effect, we can truncate the expression after the first solid. For example, `(car (append n (cons b m)))` and `(car (append n (cons b inil)))` have the same normal forms, because the car must come either from `n` or, if `n` is empty, from `b`. This truncation can be a bit tricky since each cdr expression discards one solid: `(car (cdr (append n (cons b m))))` may really need `m`, but `(car (cdr (append n (cons b (cons c m)))))` will not.

In short, the normal form for list expressions handles the append/cons identities, resolves cars and cdrs when possible, and truncates car expressions. The semantic analyzer never resorts to creating a new variable during the construction of normal expressions for lists. This does not mean, however, that normal list expressions are more accurate that normal integer expressions or normal bitvector expressions. Two list expressions which always evaluate to the same list may have different normal expressions. For example, `(cdr (cdr (append n n)))` has the normal form $\text{cdr}(\text{cdr}(n \cdot n))$ and `(cdr (append (cdr n) n))` has the normal form $\text{cdr}(\text{cdr}(n) \cdot n)$, but they always evaluate to the same list for any assignment to n.

Chapter 5

Repairing Violations

5.1 Overview

The methods developed in the previous chapter can be used to find conservative thinnings. A trace graph with a conservative thinning serves as a *thinning example*: an example of wasted computation that programs ought to avoid. Next, we need a method for "learning the lesson" of a thinning example. That is, we need a way to modify a trace grammar with respect to a thinning example, to make sure that no graph derivable in the grammar makes the mistake illustrated by the example. I call this the *grammar thinning problem* [Web92c].

That there is a correspondence between trace grammars and CFG's has already been noted. My original plan of attack for the grammar thinning problem was to make use of that correspondence. I planned to identify the relevant CFG thinning problem, solve it, and then find the corresponding solution to the trace grammar thinning problem. I imagined that the CFG thinning problem would be relatively easy to solve, and that the difficult part would be lifting any solution to the realm of trace grammars. This turned out to be incorrect: the grammar thinning problem for CFG's is quite a rich problem and is still open, but the approximate solution detailed in this chapter applies to trace grammars with little modification. It seems that much of the complexity of the trace grammar thinning problem is captured by the CFG thinning problem.

In the CFG thinning problem, a thinning example is a string δ with one or more symbols marked. To thin a language with respect to this δ is to edit each word to reflect the observation that the marked symbols of δ are unnecessary and should be omitted. When δ 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 $L(H) = L_\delta(G)$. This procedure is developed in Section 5.2.

But it is only a special-case solution. There are fully terminal thinning examples that don't satisfy the condition, and, moreover, thinning examples may include nonterminals. We'll extend our definition of $L_\delta(G)$ to allow nonterminals in δ, by specifying a thinning procedure to be performed on each parse tree generated by G, rather than on each string in $L(G)$. This yields the general CFG thinning problem: for any grammar G and thinning example δ, find a grammar H for which $L(H) = L_\delta(G)$. It is an open problem; it is not even known whether $L_\delta(G)$ is necessarily context free.

Our ultimate motivation is the optimization of trace grammars, so we don't need to hold out for an exact solution. An approximate one, one that eliminates some but not necessarily all instances of the thinning example, will do. One approximate solution 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 Δ, we can use this method to get an approximate solution to the general problem.

Unfortunately, this approximate solution isn't strong enough for some of the optimization problems Thinner should be able to solve. In Section 5.5 we will examine a more powerful method of constructing an approximate solution called the *method of kernels*. This method is the one actually implemented in Thinner. Section 5.6 shows an example of how the method of kernels can be applied to trace grammars. (The details of this application will be covered in the next chapter, which treats the implementation of Thinner.)

5.2 Fully terminal CFG thinning

In the context-free grammar version of the thinning problem we're 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'll 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[1] 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. 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 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 $L(H) = L_\delta(G)$. 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 δ?

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

2. Can we give an algorithm that takes a CFG G and a thinning example δ 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 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 $\underline{aa}bc$ 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). As far as I can tell, the language $M(\delta)$ is context free for all δ, but this is unproven.

Conjecture 5.1 *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 I have a solution to the CFG thinning problem.

Theorem 5.2 *If $M(\delta)$ is a regular language then $L_\delta(G)$ is context free for any context-free grammar 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_δ as follows. P_δ 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_δ 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_δ are those in which both P and F are in accepting states.

An input string w is accepted by P_δ 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 δ. So P_δ accepts w if and only if $w \in L_\delta(G)$. □

5.2.1 Thinning by FSM

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.) There is always the roundabout method suggested by the proof: construct a pushdown automaton P for G, use it and the finite-state automaton F for $M(\delta)$ to construct P_δ as described above, and then use the classical construction [HU79] to make a grammar H that generates the language accepted by P_δ. 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 5.2.

Alternate Proof of Theorem 5.2: Suppose G is a CFG and δ 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 $\underline{x}$. 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. □

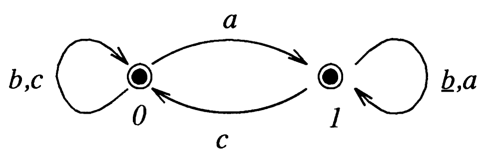
This proof suggests a method for transforming grammars—a method I have implemented. The tricky part is the construction of a CFG for the intersection of two languages, one a context-free language given by a CFG and the other a regular language given by an FSM. Salomaa [Sal73] gives a treatment of this problem. The general idea is to specialize each nonterminal X in the grammar into a variety of nonterminals $X_{i,j}$, 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 nonterminals. It also simplifies the resulting grammar a bit, getting rid of self-productions, duplicate nonterminals (i.e. those with identical productions), and nonterminals with single productions. Figure 5.1 shows a grammar thinning performed by this implementation.

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 5.2 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's regular whenever δ has exactly one marked symbol, and it's regular even for some fairly tricky δ's like $\delta = \underline{x}y\underline{x}y$ and $\delta = \underline{a}b\underline{x}a\underline{b}$ (which were used above in an exercise for the reader).

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

$S \rightarrow AB$
$A \rightarrow aA \mid a \mid c$
$B \rightarrow bB \mid b$

Initial grammar G



FSM for $M(\delta)$, $\delta = a\underline{b}$. Start state is 0 and both states accept.

$S \rightarrow S_{0,0} \mid S_{0,1}$
$S_{0,0} \rightarrow A_{0,0}B_{0,0}$
$S_{0,1} \rightarrow A_{0,1}B_{1,1}$
$A_{0,0} \rightarrow aA_{1,0} \mid c$
$A_{0,1} \rightarrow aA_{1,1} \mid a$
$A_{1,0} \rightarrow aA_{1,0} \mid c$
$A_{1,1} \rightarrow aA_{1,1} \mid a$
$B_{0,0} \rightarrow bB_{0,0} \mid b$
$B_{1,1} \rightarrow B_{1,1} \mid \epsilon$

Thinned grammar, before simplification

$S \rightarrow A_{x,0}B_{0,0} \mid A_{x,1}$
$A_{x,0} \rightarrow aA_{x,0} \mid c$
$A_{x,1} \rightarrow aA_{x,1} \mid a$
$B_{0,0} \rightarrow bB_{0,0} \mid b$

Final thinned grammar

Figure 5.1: Construction of a grammar for $L_\delta(G)$

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

Figure 5.2: Some thinning examples classified

Conjecture 5.3 *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 doesn't work the other way, as shown by the case $\delta = \underline{a}a\underline{a}$. I believe that a strengthened form of this condition is both necessary and sufficient.

5.3 General CFG thinning

The trouble with the terminal solution presented above is that it won't 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 doesn't 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 an *instance* of δ in a tree T to be a string of nodes of T that matches δ and occurs on some search frontier in 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 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 δ, 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 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're not editing strings, the question of whether the string language $M(\delta)$ is regular seems irrelevant.

5.4 The thinning-set method

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.

procedure SetThinTree(T, Δ);

  T is a parse tree which is *edited in place*;

  Δ is an ordered set of trees, each with at least one marked node;

begin

  while some member of Δ is a prefix of T

    begin

      Q := the first member of Δ which is a prefix of T;

      delete from T those subtrees matched to marked nodes in Q

    end

end;

function WeakThinTree(T, Δ);

  T is a parse tree;

  Δ is an ordered set of trees, each with at least one marked node;

  WeakThinTree(T, Δ) is a tree, a pruned version of T;

begin

  for each subtree Q of T in a topological order

    SetThinTree(Q, Δ);

  return T

end;

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

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

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 Δ 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 δ 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 5.4 *$L_\Delta(G)$ is context free for any thinning set Δ 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 Δ-*deep* tree as one in which any nonterminal which is not expanded has distance exactly n from the root, where n is the maximum height of any tree in Δ. (Note that a Δ-deep tree may have paths of length $< n$ as long as they end at terminals.)

procedure ExtendThinning(R, G, Δ);

R is a parse tree which is *edited in place*;

G is a context-free grammar;

Δ is an ordered set of trees, each with at least one marked node;

begin

if R is less than Δ-deep **then** make it Δ-deep by extending it using grammar G;

if some member of Δ is a prefix of R

 then

  begin

   $Q :=$ the first member of Δ which is a prefix of R;

   delete from R those subtrees matched to marked nodes in Q;

```
7            ExtendThinning(R, G, Δ)
8        end
9    else
10       for each nonterminal child Q of R in order
11           ExtendThinning(Q, G, Δ)
end;
```

The idea is to apply this procedure using the start symbol of G as the tree to extend: ExtendThinning(S, G, Δ). It isn't a fully specified procedure, since it doesn't say how R should be extended using grammar G at line 1. The idea is that ExtendThinning can generate any tree grammar G can generate, but thins the tree as it generates it. The recursive call at line 7 thins the developing tree iteratively, just as WeakThinTree does when it calls SetThinTree. The recursive call at line 11 visits each subtree in a topological order, just as WeakThinTree does. (As long as it's topological, the actual order is irrelevant, since the subtrees are thinned independently.) The only thing different about 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 ExtendThinning(S, G, Δ) is the same as the language of trees generated from G in the normal way and then thinned by WeakThinTree using the thinning set Δ.

Observe that ExtendThinning(S, G, Δ) never results in an ExtendThinning call with a tree that is more than Δ-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 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 Δ. Thus there are only finitely many different parameters R on which

ExtendThinning will call itself.

Now we construct a CFG H that mimics the behavior of ExtendThinning. The terminal alphabet of H will be the same as the terminal alphabet of G, but the nonterminal 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 ExtendThinning will call itself. For each nonterminal symbol Z_R in H the productions are determined as follows. Let X be the set of Δ-deep expansions of R using grammar G; this will be just $\{R\}$ if R is already Δ-deep. (X corresponds to the set of possible extensions generated at line 1.) For each $S \in X$ with a prefix in Δ there is a production $Z_R \rightarrow Z_Q$, where Q is the result of thinning S using the first prefix of S in Δ. (Q corresponds to the thinning computed at lines 5-7.) For each $S \in X$ with no prefix in Δ, let Q_i be the ith 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 Z_{Q_i} if Q_i is not a terminal node. (This corresponds to recursively visiting each nonterminal 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, Δ). 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 δ and a grammar G, I generate a thinning set Δ of all the derivations in G that generate δ 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 nonterminals 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 Δ-deep trees, as ExtendThinning does at line 1; instead, it extends a tree only until the next thinning step is determined. Figure 5.3 shows a grammar thinning as generated by this implementation.

5.5 The method of kernels

Later we will see that the example of Figure 5.3 is a CFG analog of the Fibonacci reformulation for trace grammars. Since the thinning-set method solves this example, it is tempting to conclude that it is a sufficiently powerful method for Thinner (assuming that it can be lifted to the trace grammar thinning problem). But a closer examination shows that this is incorrect: the thinning-set method has a critical weakness.

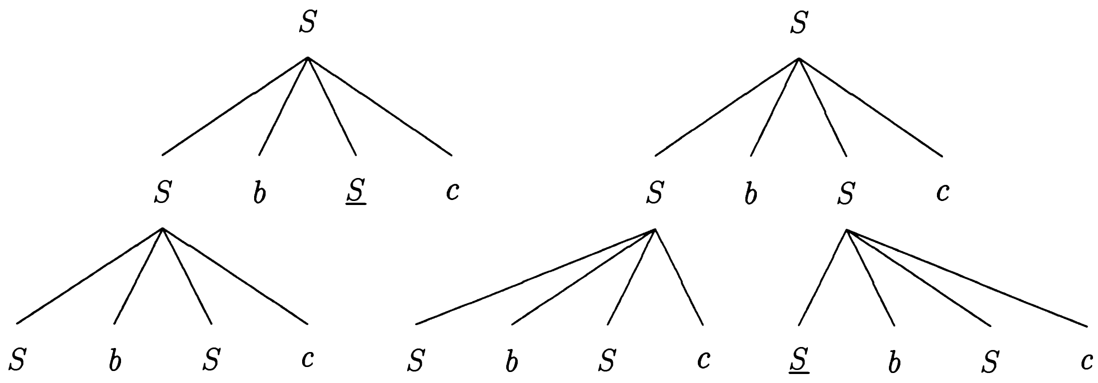
5.5.1 A weakness of the thinning-set method

That weakness lies in the fixed maximum height of trees in the thinning set. Consider the problem of thinning the same grammar $S \rightarrow SbSc \mid a$ with respect to a different thinning example, $\delta = b\underline{Sb}$. Figure 5.4 shows that a parse tree in this grammar may require thinnings that span arbitrarily tall subtrees: the distance to the common ancestor of the nodes involved in a thinning is unbounded. Any fixed thinning set will repair only a few instances of the thinning example, leaving all "taller" instances uncorrected.

This example, as it happens, is a CFG analog of loop invariant removal for trace grammars: the distance in a derivation tree between the first instance of the loop-invariant expression and some repetition of it is unbounded. It's clear the thinning-set method is inadequate: we need, if not an exact solution, then at least an approximate solution that avoids the fixed-height limitation.

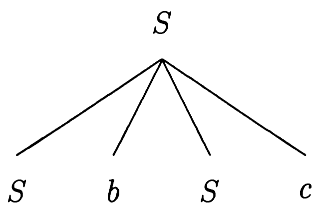
$$S \rightarrow SbSc \mid a$$

Initial grammar G



Thinning set Δ: all trees of depth $d \leq 2$

that derive $\delta = cb\underline{S}$



Tree T used in thinned grammar

$$S \rightarrow Z_T \mid a$$

$$Z_T \rightarrow Z_T bc \mid abSc$$

Thinned grammar

Figure 5.3: Construction of a grammar for $L_\Delta(G)$

$S \rightarrow SbSc \mid a$

$\delta = b\underline{Sb}$

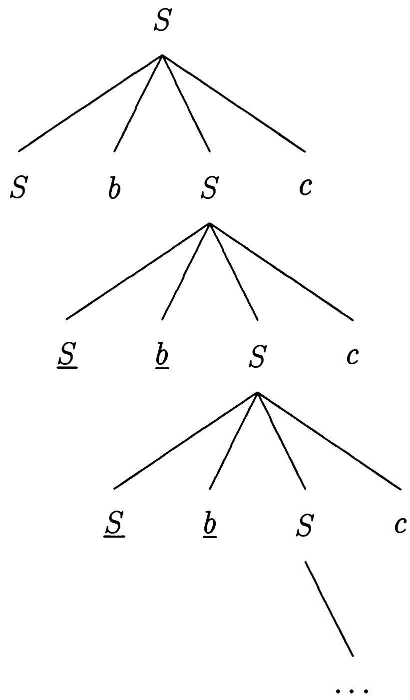


Figure 5.4: A problem that defeats the thinning-set method

(It is important to point out that although this problem escapes the thinning-set method, it is still solvable; as for all examples of which I am aware, one can manually construct a grammar H for which $L(H) = L_\delta(G)$. In this case we need only observe that when the production $S \to SbSc$ is used in a parse tree, the possible terminal fringes left below the second S are exactly the regular language ac^*. So H could have productions $S \to SbAc|a$ and $A \to Ac|a$.)

5.5.2 Outline of the method of kernels

The purpose of the nonterminals Z_R in the thinning-set construction is to hold together a piece of the derivation until it becomes clear whether or not that piece yields an instance of the thinning example. The next method will use the same general idea; but instead of creating new nonterminals corresponding to partial parse trees, we'll create new nonterminals corresponding to strings of symbols called *kernels*.

Consider the thinning example $\delta = b\underline{Sb}$: how can this arise as a substring of some sentential form in the grammar $S \to SbSc \mid a$? 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 $S \to SbSc$. The string bS is therefore a *kernel* of δ in the grammar: it's 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 $S \to SbSc$.[2]

More formally, we define a kernel as follows:

Definition 5.1 *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*

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

$\phi \Rightarrow^*_G \alpha\delta\beta$ 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).

Once a kernel arises in a derivation, we're sunk: the nonterminal symbols in a kernel expand independently, and depending on these expansions we may or may not end up with an instance of δ. 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

  G' := a new, empty grammar;

  for each production $A \to \alpha$ in G **do**

    Kernelize$(A, \alpha, G, G', \delta, K)$;

  return G'

end;

procedure Kernelize$(A, \alpha, G, G', \delta, K)$;

  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;

K is a set of kernels for δ in G;

begin

1 $\alpha :=$ ThinString(α, δ);

2 **for** each kernel $k \in K$ **do**

3   **for** each x, y such that $\alpha[x..y] = k$ **do**

4   note that α cannot safely be broken between x and y;

5 break α into groups wherever it can safely be broken;

6 break further if necessary to keep group size below some constant c;

7 **for** each group $\beta = \alpha[x..y]$ with $|\beta| \geq 2$ **do**

8   **begin**

9   replace symbols x through y of 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$(X_\beta, \beta', G, G', \delta, K)$

15   **end**

16   **end**;

17 add production $A \rightarrow \alpha$ 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's 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 $b\underline{S}\underline{b}$. 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 $L(G') = L_\delta(G)$.

5.5.3 Shortcomings of the method of kernels

Why isn't the grammar returned by ThinByKernels 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:

$$S \rightarrow SS \mid a \mid b$$

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:

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

This is thinner than the original grammar, but it isn't an exact solution. (An exact solution can be obtained in this case using the FSM method of Section 5.2.1.) Without the group size bound ThinByKernels would diverge, trying to construct an infinite grammar like this:

$$S \rightarrow X_{SS} \mid a \mid b$$

$$X_{SS} \rightarrow X_{SSS} \mid X_{aS} \mid bS$$

$$X_{SSS} \rightarrow X_{SSSS} \mid X_{aSS} \mid bX_{SS}$$

$$X_{SSSS} \rightarrow \ldots$$

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're faced with the following grammar G:

$$S \rightarrow AB$$

$$A \rightarrow aAb \mid x$$

$$B \rightarrow bBa \mid y$$

and suppose we want to thin it with respect to the thinning example $\delta = \underline{A}bB$. 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's a grammar that generates $L_\delta(G)$:

$$S \rightarrow xy \mid aby \mid bBa$$

$$B \rightarrow bBa \mid y$$

But the method of kernels fails on this problem. Clearly AB is a kernel for δ in G: it's 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 isn't as thin as possible either.

The problem is that we can't 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.

5.6 The method of kernels for trace grammars

The method of kernels has its limitations, but it is still the most general solution known for the CFG thinning problem. As we will see, it carries over 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 the next chapter; for now, we will sketch the idea by applying the method of kernels to analogous CFG and trace grammar problems, side by side.

We'll start with this CFG G:

$S \rightarrow SbSc \mid a$

and we'll 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'll divide the operation of ThinByKernels into five steps:

1. Thin and group—in this step, the right-hand side of $S \to SbSc$ is thinned (which has no effect) and grouped into kernels, leaving $S \to SbS \cdot c$. ($S \to a$, 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 $S \to X_{SbS}ca$.

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 $X_{SbS} \to SbScbS$ is thinned (which reduces it to $SbScb$) and grouped into kernels, leaving $S \to SbS \cdot cb$. ($X_{SbS} \to abS$, 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 $X_{SbS} \to X_{SbS}cb$.

The resulting grammar is:

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

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

These same five steps are echoed in the optimization of the exponential-time Fibonacci function. Figure 5.5 shows the appropriate trace grammar thinning problem: note that, as in the CFG case, we have a single kernel; that kernel contains two nonterminals; and when one of them (the one that computes `(fib (- n 1))`) is expanded, the resulting graph contains the thinning example.

In step 1, we thin and group the right-hand side of the second production for `fib`. (The other production is carried into the new grammar unchanged.) Thinning the right-hand side has no effect since the thinning example doesn't occur there. The right-hand side graph after grouping is shown in Figure 5.6.

Step 2 replaces the group shown in Figure 5.6 with a new nonterminal X_β. This yields the productions shown in Figure 5.7.

We choose a nonterminal to expand and expand it in step 3. 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, and it's the one we'll expand. There are two productions for `fib` in the original grammar, so this expansion yields two possible right-hand sides for X_β, as shown in Figure 5.8.

In step 4, the graphs shown in Figure 5.8 are thinned and grouped. This time thinning does have an effect, 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 5.8 is carried into the new grammar unchanged, and the resulting productions for X_β are shown in Figure 5.9.

Finally, in step 5, we replace the group β with the nonterminal X_β. The final grammar is shown in Figure 5.10. It is an exact solution: a thinner grammar that never generates a graph that contains the thinning example. This is essentially how Thinner optimizes the exponential-time Fibonacci function, as detailed in Chapter 6. The grammar shown in Figure 5.10 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.

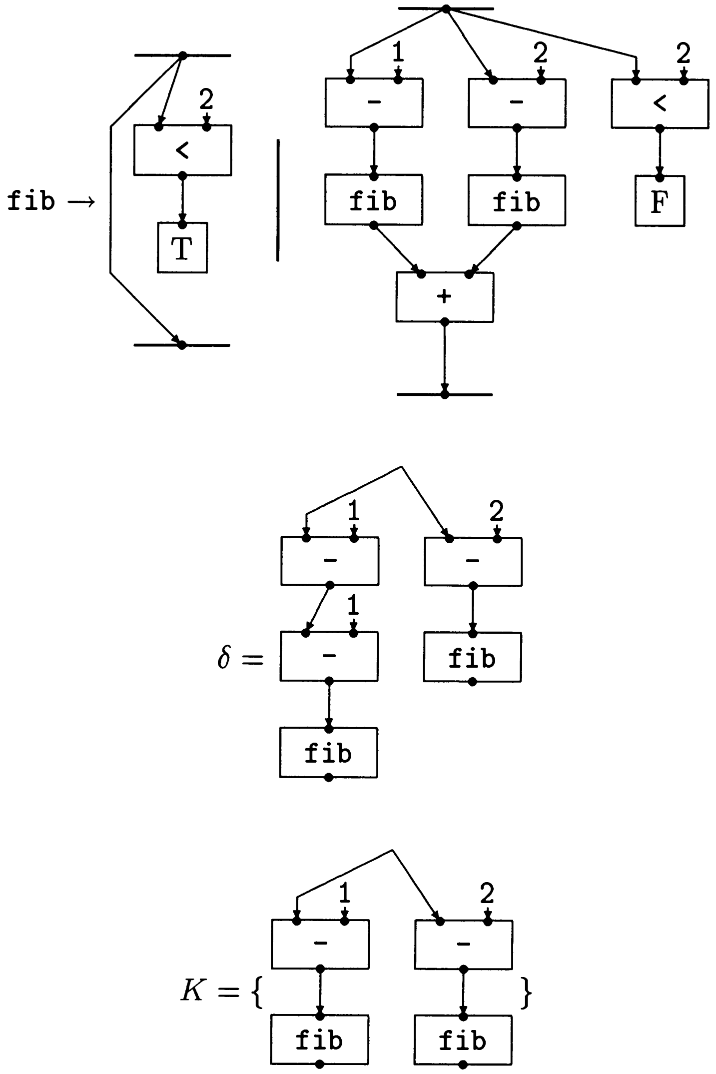


Figure 5.5: A trace grammar thinning problem

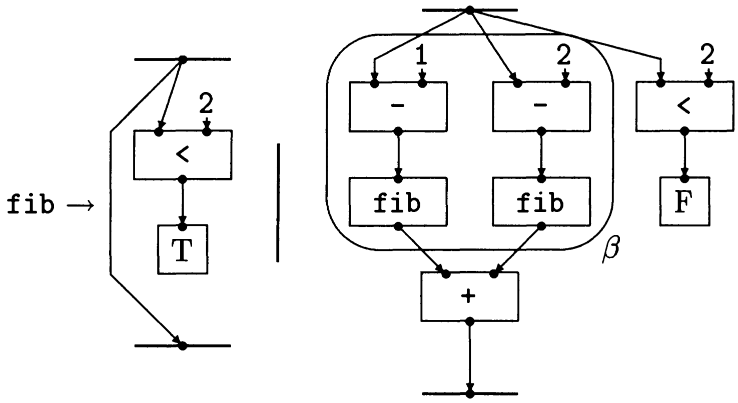


Figure 5.6: Productions for `fib` after step 1, showing group β

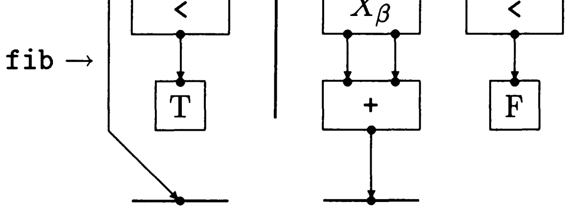


Figure 5.7: Productions for `fib` after step 2

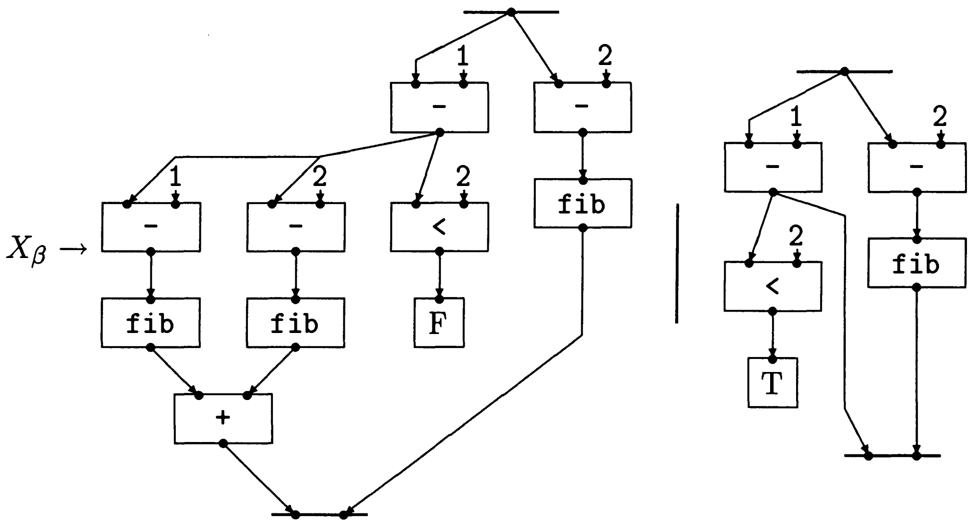


Figure 5.8: Productions for X_β generated in step 3

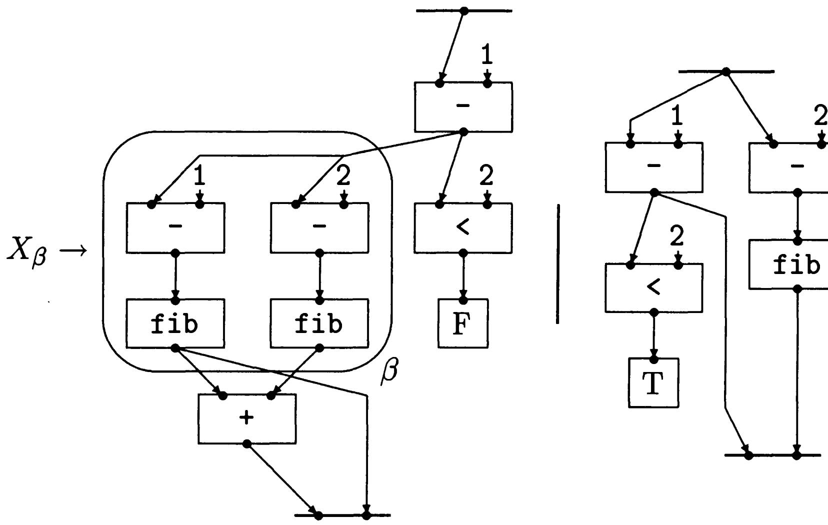


Figure 5.9: Productions for X_β after step 4

In particular, it corresponds to a linear-time, rather than an exponential-time, function.[3]

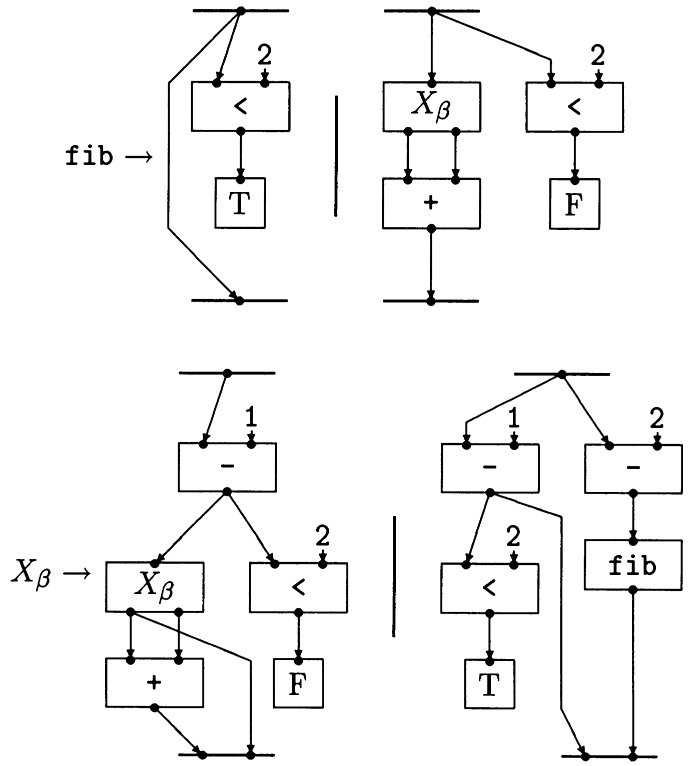


Figure 5.10: Final grammar after step 5

[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 [Roh89].

Chapter 6

Thinner

> The lower panel had not slid aside to reveal FINANCIAL MATTERS WILL SOON IMPROVE or OLD FRIENDS WILL VISIT or DO NOT MAKE IMPORTANT DECISION HASTILY.
>
> It had revealed a single black word: "THINNER."
>
> — *Stephen King*, Thinner

6.1 Overview

We now turn from neat to scruffy—from theory to experiment. Earlier chapters have provided the following tools:

- the trace grammar representation (Chapter 2);
- a formal semantics giving denotations of TL programs in terms of trace grammars (Chapter 3);
- the relational constraint inference technique (Chapter 4); and
- the method of kernels for thinning CFG's (Chapter 5).

This chapter describes the synthesis of these tools in Thinner, an optimizer for TL programs.

Thinner is a Common Lisp system of about 15,000 lines, implemented in Lucid Common Lisp on a Sun-4 workstation.[1] Figure 6.1 shows the major components of Thinner. Thinner is a source-to-source optimizer: a *compiler* converts the input program into a trace grammar, and a *decompiler* converts the final trace grammar back into a program in the source language. In between, Thinner explores the trace grammar, looking for opportunities for optimization. It is an iterative process, and Thinner takes as much time as the user gives it. The more time it has, the more optimizations it may discover.

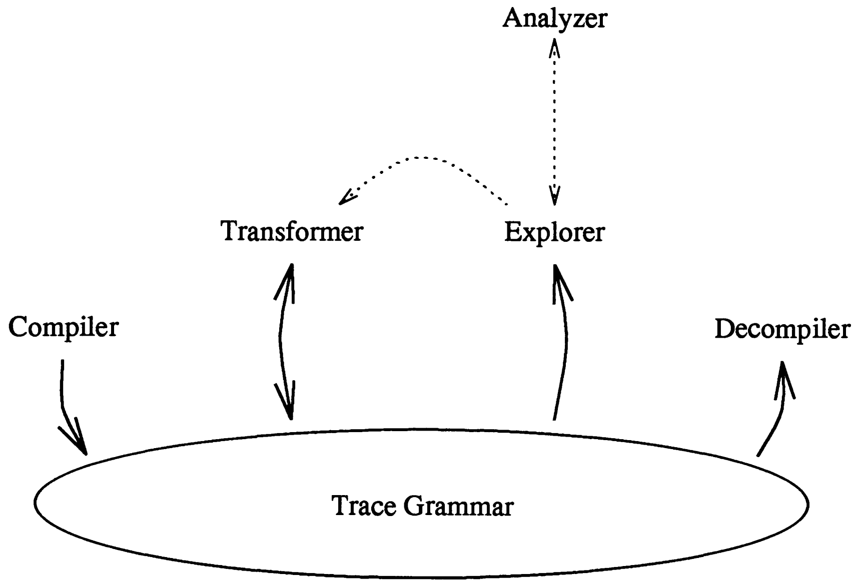


Figure 6.1: Structural overview of Thinner

The grammar *transformer* works with a thinning example: a trace graph that has been shown to include unnecessary computation. (This thinning example may include nonterminals.) The transformer uses the method of kernels to reformulate the grammar. Its goal is to alter the grammar so that the language it generates

[1]Lucid and Lucid Common Lisp are trademarks of Lucid, Inc. Sun-4 is a trademark of Sun Microsystems, Inc.

never makes the mistake shown in the thinning example.

Where does the thinning example come from? A component called the *explorer* looks for unnecessary computations committed by the trace grammar. It looks at the right-hand sides of productions in the grammar, and it unfolds them along possible derivations. In effect, it examines individual partial paths through the program, embodying each such path in a trace graph.

The dotted lines in Figure 6.1 represent paths along which individual trace graphs are passed. The explorer calls on the *analyzer* to reason about individual trace graphs, to determine whether they contain any unnecessary computation—that is, to determine whether they compute redundant or unused values. The analyzer uses the relational constraint method to identify equivalences among objects in a trace graph. It annotates the trace graph with equivalence information and returns it to the explorer. If there are suitable equivalences the graph is then passed to the transformer as a thinning example. Thinner is a "global" optimizer in the sense that the explorer examines interprocedural slices of the computation looking for thinning examples, and the transformer tries to correct all instances of a given example. But the analyzer reasons locally about each trace graph the explorer gives it.

This chapter 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 chapters into a practical system. It includes some illustrative selections of code from Thinner, and a selection of annotated transcripts showing Thinner at work.

6.2 Thinner fundamentals

Here are some details that may help the reader to understand the code examples that follow.

6.2.1 Memory management

Thinner follows a strict memory-management regimen: it explicitly frees all pieces of memory it allocates, including cons cells. This is rather compulsive for a Lisp system since garbage collection will find any lost objects. But I wanted to gather statistics on Thinner's use of time and space without interference from garbage collection. All data objects are allocated and freed using macros that manipulate a list of free objects of the appropriate type.

6.2.2 Doubly-linked circular lists

When Thinner needs a list data structure that supports random insertions or deletions, it uses a doubly linked circular list structure called a `dlist`. Many of the higher-level data structures of Thinner—graphs, vertices, terminal and nonterminal functions, and so on—start with the `dlist` structure and can be manipulated in circular lists.

6.2.3 Iteration macros

Thinner uses a macro package for iteration. These macros were originally written by Professor Alberto Segre of Cornell University; he remarks that they were inspired by the Interlisp FOR clisp macro. I modified them only slightly for the Thinner project. Keywords beginning with a colon specify various iteration options.

6.3 The compiler

The structure of the compiler follows the structure of the denotational semantics of TL: the compiler builds up the trace graph for an expression by composing the trace graphs for subexpressions.

The compiler converts a TL function into a nonterminal with a collection of trace graphs for productions. A nonterminal is represented by a `function` structure as shown in Figure 6.2.

```
;; A function structure is a terminal or nonterminal "symbol" in a
;; trace grammar.  All vc vertices with that function are linked to
;; the function structure through (fref-function (vertex-fref v));
;; the frefs are collected in a dlist, so we can find all uses of a
;; given function in the grammar.

(defstruct (function
             (:include dlist)
             (:print-function print-function))
  id        ; unique id
  class     ; term or nonterm
  name      ; name of this function
  tin       ; vector of input types
  tout      ; vector of output types
  init-fcon ; a function to initialize functional constraints
  nform     ; a function to compute normal forms for outputs
  refs      ; dlist of frefs to this function
  graphs    ; nonterms: dlist of rhs graphs for this function
  transp    ; nonterms: list of lists of relations
  dtrees    ; nonterms: list of decision trees
  n1
  )
```

Figure 6.2: The `function` structure

Terminal functions (that is, primitive functions in TL) are represented using the same structure. For each TL primitive there is a predefined `function`, like the one for addition shown in Figure 6.3. The `init-fcon` field is a function that initializes the functional constraints established by a nonterminal vertex for addition. The `nform` field is a function that computes the normal form for the output, assuming that the normal forms for the inputs are already available.

```
;; *function-plus* is the terminal function for addition.

(defconstant *function-plus*
  (make-function
   :id (incf *function-id*)
   :class 'term
   :name '+
   :refs (new-dlist)
   :tin (vector *i* *i*) :tout (vector *i*)
   :init-fcon #'(lambda (v g)
                  (let* ((refs (new-vector 6))
                         (fcon (new-fcon #'plus-constraint refs))
                         (a (inec v 0))
                         (b (inec v 1))
                         (c (outec v 0))
                         (one (graph-one g)))
                    (add-pairref (find-pair a b g *i*) fcon 0 t g)
                    (add-pairref (find-pair b c g *i*) fcon 1 t g)
                    (add-pairref (find-pair a c g *i*) fcon 2 t g)
                    (add-pairref (find-pair a one g *i*) fcon 3 t g)
                    (add-pairref (find-pair b one g *i*) fcon 4 t g)
                    (add-pairref (find-pair c one g *i*) fcon 5 t g)
                    (insert-dlist (graph-fcons g) fcon)
                    (ftryst fcon g)))
   :nform #'(lambda (v i)
              (declare (ignore i))
              (sum-poly (inform v 0 nil) (inform v 1 nil)))))
```

Figure 6.3: The `function` for addition

Each nonterminal function has a list of trace graphs which are the right-hand sides of productions for that nonterminal in the grammar. The **graph** structure, shown in part in Figure 6.4, records the parts of a trace graph much as you might expect from Definition 2.2. Each vertex in the trace graph is represented by a `vertex` structure. For each vertex output there is an `ob` structure; a graph contains, in total, one of these `ob` structures for each vertex output and one for each constant used in the graph. The `ob`'s are the focus of inference. The analyzer tries to collect them into equivalence classes, since objects in the same equivalence class can serve as bypasses for each other during thinning.

TL contains a variety of overloaded operators, but the compiler must decide on a unique type for each object in the graph. TL includes a way to declare variable types, but Thinner performs some type inference (in the style of ML) so that type declarations are often unnecessary. This is just a convenience for the user; when Thinner decompiles, it always includes full type declarations.

6.4 The decompiler

At any time the user of Thinner can ask to see a decompiled version of the current grammar. The grammar transformer guarantees that the current grammar is always normal (Definition 2.19), 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 2.11) for the set of right-hand sides for that nonterminal, so the first step is really the computation of a decision tree. Thinner does this by matching, from the inputs down, all graphs in the set as far as possible, splitting the set on a matched predicate (there must be a matched one since the set is normal), and then proceeding recursively on the two smaller sets. There may be more than one decision tree, and Thinner will find them all. Each decision tree would yield a slightly different (though grammatically and functionally equivalent) decompilation; Thinner

```
(defstruct (graph
             (:include dlist)
             (:print-function print-graph))
  function  ; function in which this graph occurs
  vi        ; a vector of input vertices
  vo        ; the output vertex
  vc        ; a dlist of computational vertices
  vt        ; a dlist of true predicates
  vf        ; a dlist of false predicates
  ...
  )


(defstruct (vertex
             (:include dlist)
             (:print-function print-vertex))
  id        ; unique id for debugging
  class     ; one of 'vi, 'vo, 'vc, 'vt or 'vf
  vin       ; vector of obrefs to source object
  vout      ; vector of output obs
  fref      ; fref to function of this vertex
  graph     ; graph in which this vertex occurs
  ...
  )


(defstruct (ob
             (:include dlist)
             (:print-function print-ob))
  type      ; object type
  vertex    ; if vertex output, the vertex defining this object
  index     ; if vertex output, the index in vout of the vertex
  name      ; print name: constant value, variable name or gensym
  ec        ; equivalence class (ec structure) of object
  refs      ; dlist of obrefs
  ...
  )
```

Figure 6.4: Graphs, vertices and objects

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 6.7.)

Decision tree in hand, Thinner proceeds to decompile the set of right-hand sides for the nonterminal. All graphs in the set match, up to the predicates at the first branch in the decision tree. Thinner decompiles this matching part (this is just like decompiling an individual trace graph without any predicates, which is straightforward). That first branch in the decision tree splits the set of trace graphs in two. Thinner emits an `if`, whose test is the predicate input and whose two arms are obtained by recursively decompiling the two parts of the set of trace graphs, guided by the two subtrees of the decision tree.

6.5 The explorer

The explorer generates graphs to be analyzed as potential thinning examples. Using depth-first iterative deepening, it explores the space of graphs that can be reached from nonterminals in the grammar, where each "move" involves selecting and unfolding one nonterminal in the current graph. This isn't systematic: it will sometimes develop the same graph along more than one path. It isn't particularly clever either.

Only one heuristic is employed to guide the explorer toward significant thinning examples: it 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.

```
(defun dfids ()
 (let ((example nil) ; thinning example
       thinnings ; dlist of thinnings
       kernels) ; list of kernel graphs

   ;; Sort productions by number of nonterminals.  This controls
   ;; the order of the search, making us explore base cases last.

   (order-graphs)

   ;; Find the first example by iterative deepening.

   (for depth :from 0 :until example
     :do (multiple-value-setq (example thinnings kernels)
           (first-example-in depth)))

   ;; Thin the grammar using the method of kernels.

   (thin-by-kernels example thinnings kernels)

   ;; Clean up the example, the thinnings, and all kernel graphs.

   (cleanup-graph example)
   (cleanup-thinnings thinnings)
   (free-dlist thinnings)
   (while kernels
     :do (let ((g (mpop kernels)))
           (unless (eq g example)
             (cleanup-graph g)))))

 ;; Finally, clean up any unused parameters, returned values, and
 ;; functions, in any parts of the grammar we just fabricated.

 (cleanup-interfaces)
 nil)
```

Figure 6.5: Top-level thinning function

Figure 6.5 shows the top-level thinning function in Thinner. This function performs the iterative deepening search and invokes the method of kernels when an example is found.

6.6 The analyzer

The analyzer examines each trace graph proposed by the explorer, to discover whether it can be thinned conservatively. If all of the values computed by some vertex in the graph are unused, no further analysis is necessary. That vertex can be removed without bypass edges, so the graph can be used as a thinning example. In general, however, more analysis is necessary: a deeper, semantically aware analysis that can detect when two vertex outputs are interchangeable (or when one is constant) for all executions of the graph. For this purpose Thinner uses the method of kernels and the computation and comparison of normal forms for expressions, which were developed in Chapter 4.

The analyzer decorates a trace graph with additional information. This information remains attached to the graph until something happens to invalidate it (that is, until the graph is altered). The most significant part of this information is a collection of equivalence classes of objects. Each equivalence class contains one or more vertex outputs from the graph; and if the equivalence class is known to include a constant, that constant is recorded. Thinner uses these equivalence classes to decide whether there is a conservative thinning for the graph.

6.6.1 Representing relations

The vocabulary of relations for each type is represented with an unsigned 29-bit word—remember that there are 29 primitive integer relations, and fewer primitive relations for the other types.[2] Figure 6.6 shows some relevant definitions from Thinner. Note the similarity between the list of primitive integer relations, `*primitives.i*`, and the definitions in Table 4.1.

6.6.2 Constructing constraint sets

The method of constraints needs a constraint set for every primitive TL function. These constraint sets are constructed with mechanical assistance. They are constructed at compile time, and when Thinner runs they are available as constants. (Because this construction takes place at compile time, these functions do not follow Thinner's usual memory-management regimen.)

Figure 6.7 shows the Thinner function that returns a constraint set for `cdr`. The intricacy of this example is about average for constraint set constructors: some, like addition, require more finesse; others, like the boolean operators, are trivial. The constraint set constructors generate sets of primitive relations. As noted in Section 4.2.2, these are reduced automatically to minimal sets of cubes before being used by Thinner.

If errors creep into a constraint set, Thinner will make errors in inference. Because the method of constraints yields assertions rather than proofs, such errors are very difficult to trace. To reduce the risk of this kind of error, I built functions to test each of the constraint sets. Figure 6.8 shows the test for the `cdr` constraint set constructed by the previous example.

[2]This representation allows the Lucid compiler to generate in-line instructions for operations on relations. With less care it is easy to end up with a library call for every logical-and and logical-or.

```
;; Type rel is a relation (in any vocabulary).

(deftype rel () '(unsigned-byte 29))


;; (conj-rel a b ...) takes any two or more relations and returns
;; their conjunction.  The result is garbage unless the relations
;; are from the same vocabulary.

(defmacro conj-rel (&rest rels)
  `(the rel (logand ,.rels)))


;; *primitives.i* is a list of the symbols for primitive integer
;; relations.

(defconstant *primitives.i*
  '(--<2+ --<1 0-<1 0-<2+ -+<2+ -+<1 -+=0
    +-<1 +-<2+ 0+<2+ 0+<1 ++<2+ ++<1
    --=0 00=0 ++=0
    -->2+ -->1 -0>1 -0>2+ +->2+ +->1 +-=0
    -+>1 -+>2+ +0>2+ +0>1 ++>2+ ++>1))


;; Define each symbol for a primitive integer relation, giving each
;; one a bit of its own.

(for name :in *primitives.i*
  :as bitno :from (1- (length *primitives.i*)) :by -1
  :do (eval `(defconstant
              ,name
              (ash 1 ,bitno))))
```

Figure 6.6: Definitions for relations on integers

```
;; (list-cdr-base) -- This returns a list of the possible triples of
;; primitive list relations (a::b a::nil b::nil) given that b is the
;; cdr of a.

(defun list-cdr-base ()
  (flet
      ((mklrel (s c mag)
         (eval (intern (concatenate
                        'string (if s "S" "") (if c "C" "")
                        (case mag
                          (-2 "<2+") (-1 "<1") (0 "=0")
                          (1 ">1") (2 ">2+")))))))
    (for acn :in '(t nil) ; car a = car nil?
      :join
      (for amagn :in (if (not acn) '(1 2) '(0 1 2))
        :join
        (for bmagn :in (if (<= amagn 1) '(0) '(1 2))
          :join
          (for bcn :in (if (zerop bmagn) '(t) '(t nil))
            :join
            (for acb :in (cond ; car a = car b?
                           ((or (and acn bcn) (zerop amagn)) '(t))
                           ((not (or acn bcn)) '(t nil))
                           (t '(nil)))
              :collect
              (list (mklrel t acb (if (zerop amagn) 0 1))
                    (mklrel t acn amagn)
                    (mklrel t bcn bmagn)))))))))
```

Figure 6.7: Constructor for the cdr constraint set

```
; (test-list-cdr-base) -- Test the list of tuples produced by
; list-cdr-base.  We generate all lists of length less than 5 over a
; three-integer alphabet, and test each one by taking its cdr and
; generating the appropriate tuple of primitive relations.  If any
; tuple so generated does not appear on the list produced by
; list-cdr-base, we report it.  If any tuple on the list produced by
; list-cdr-base is not generated by this process (or if it occurs on
; the list more than once) we report that.

(defun test-list-cdr-base ()
  (labels ((3lists (len)
             (if (zerop len)
                 (list nil)
                 (for x :in (3lists (1- len))
                   :join (list (cons 0 x) (cons 1 x) (cons 2 x))))))
    (let ((lists (for i :from 0 :to 4 :join (3lists i)))
          (base (list-cdr-base))
          found)
      (for a :in lists
        :do (let* ((b (cdr a))
                   (tuple (list (constant-ilrel a b)
                                (constant-ilrel a nil)
                                (constant-ilrel b nil))))
              (cond ((member tuple found :test #'equal))
                    ((member tuple base :test #'equal)
                     (setq found (cons tuple found)))
                    (t (format t "Tuple ~S not found" tuple)
                       (format t " for ~S = (cdr ~S)~%"
                               b a)))))
      (setq base (for tuple :in base
                   :unless (member tuple found :test #'equal)
                   :collect tuple))
      (when base
        (format t "~D tuples not sought: ~S"
                (length base) base)))))
```

Figure 6.8: Testing the cdr constraint set

6.6.3 Relational constraint implementation

The relational constraint algorithm presented in Section 4.2.3 is implemented in Thinner, with several optimizations as noted here. (These optimizations do not effect the asymptotic worst-case complexity of the method, which is still $O(n^2)$ in the size of the graph, but they yield significant linear improvements.)

The constraint computation described in Section 4.2.2 was a general procedure for any size tuple of relations. In practice I found it more efficient to construct specialized a version of it for each tuple size. Figure 6.9 shows the function that computes constraints on triples of relations.

In the earlier presentation of the relational constraint algorithm, the collection of constraints waiting to be computed was treated as a set. The algorithm did not specify how the next constraint to apply was to be chosen from this set. For asymptotic worst-case analysis it makes no difference which of the waiting constraints is chosen, but in practice it makes a significant difference. Obviously it's a good idea to take big strengthening steps rather than small ones, and the choice of the next constraint to perform should reflect this:

- I experimented with choosing the next constraint in LIFO, FIFO, and random order. Treating the collection of constraints as a queue gave the best performance, while treating it as a stack gave the worst. The reason seems to be that the "older" a constraint is, the more likely it is that its pairs have changed since last it was computed.

- I experimented with giving preference to transitivity constraints, with giving preference to all other constraints, and with treating all constraints equally. Leaving transitivity constraints for last gave the best performance, while doing them first gave the worst. The reason seems to be that transitivity

```
;; (tableconstraint3 p1 p2 p3 g table) makes the constraint
;; computation for triples of pairs.  p1, p2 and p3 are the pairs; g
;; is the graph, and table is the constraint set.

(defun table-constraint3 (p1 p2 p3 g table)
  (let ((a (pair-rel p1))
        (b (pair-rel p2))
        (c (pair-rel p3))
        (ra 0) (rb 0) (rc 0)
        ta tb tc)
    (declare (type rel a b c ra rb rc ta tb tc))
    (for con :in table
      :do (setq ta (conj-rel a (the rel (first con))))
          (unless (zerop ta)
            (setq tb (conj-rel b (the rel (second con))))
            (unless (zerop tb)
              (setq tc (conj-rel c (the rel (third con))))
              (unless (zerop tc)
                (setq ra (disj-rel ra ta))
                (setq rb (disj-rel rb tb))
                (setq rc (disj-rel rc tc))))))
    (strengthen-pair p1 ra g)
    (strengthen-pair p2 rb g)
    (strengthen-pair p3 rc g)))
```

Figure 6.9: Constraining triples of relations

constraints tend to be much weaker than, for example, functional constraints, yielding strengthening in smaller steps.

Making the right choices here instead of the wrong ones improved the experimental performance of the relational constraint method by about a factor of three.

The initial installation of constraints was described in Section 4.2.3 as a function Init. Init placed all constraints from the graph in the starting collection of constraints to be computed. This makes the proof of correctness easier but turns out to be unnecessary. Transitivity, list property, and transparency constraints have no effect on tuples of top relations, the initial state of most relations in the table. With a little care these unnecessary initial constraint computations can be avoided.

Lines 9-12 in Init also install a number of redundant transitivity constraints. If both (a, b) and (b, a) participate in functional constraints, this algorithm will install transitivity constraints both for the triple (a, b, c) and for the triple (b, a, c), for every c. In fact we only need one transitivity constraint per triangle: one for each set $\{a, b, c\}$, not one for each tuple (a, b, c). This can be accomplished by ordering the objects, then installing a transitivity constraint for (a, b, c) only if $a \sqsubseteq b \sqsubseteq c$ (and, as usual, only if (a, b) or (b, a) has a functional constraint).

For simplicity of presentation the algorithm treated the collection of strongest known relations (Rel) as one big table indexed by pairs of objects. In fact, we only index using pairs of objects of the same type, and it makes more sense to maintain a separate table of relations for each type. Figure 6.10 shows the Thinner function that prints the relation for every pair of equivalence classes in the given graph. (It's only used for debugging.)

A more difficult optimization Thinner uses is to reason not about objects but about equivalence classes of objects. Whenever a relation becomes as strong as

```
;; (show-rels g) prints the current relation for every pair of ecs
;;  ing.  Inference structure must be present in the graph.  We
;; print each relation in one direction only (that is, we treat the
;; rels tables as triangular), but we check to be sure that
;; rel(i,j) = rev(rel(j,i)).

(defun show-rels (g)
  (for type :from *minfulltype* :to *maxfulltype*
    :do
    (format t "~%Type ~D:~%" type)
    (let* ((rels (svref (graph-rels g) type))
           (n (array-dimension rels 0)))
      (for i :from 0 :to (1- n)
        :do (for j :from i :to (1- n)
              :do (let ((pair (svref (svref rels i) j)))
                    (when pair
                      (let* ((ec1 (pair-ec1 pair))
                             (ec2 (pair-ec2 pair))
                             (rel (pair-rel pair))
                             (rp (pair-rev pair))
                             (rrel (pair-rel rp)))
                        (format t "    ~S " ec1)
                        (print-rel rel type)
                        (format t " ~S~%" ec2)
                        (assert (= rel (rev-rel rrel type))))))))))
```

Figure 6.10: Examining the relations for a graph

equality, Thinner merges two equivalence classes and removes a row and column from the table. This makes a big difference in the number of transitivity constraints that are computed, and helps make up for using only $O(n^2)$ of them.

6.7 The transformer

We saw in Section 5.6 that one can apply the method of kernels to trace grammars just by generalizing each step from strings to graphs. But this generalization is far easier to imagine than to implement.

6.7.1 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 to repeatedly find and thin 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. I do not know whether the subgraph matching problem for trace grammars *generated from TL programs* is actually NP-hard. I have not been able to construct a family of programs that makes Thinner's subgraph matching routines exhibit exponential-time behavior.

These are some of the graph-matching tasks Thinner performs:

- Determine whether two predicates match—that is, whether their trees of predecessors are identical.
- Find an isomorphism, if there is one, between two graphs, treating the order of the graph inputs and outputs as irrelevant.
- Find each way one graph can be a subgraph of another.
- Find each way two graphs can overlap (this is used to find kernels of a thinning example).
- Look up a graph in a hash table of graphs.

6.7.2 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 non-empty list of potential thinnings of that graph. Figure 6.11 shows the structure used in Thinner to represent an individual thinning. (The weights are not immediately significant for a thinning example; they are used, as described below, when the example is matched to some graph in the grammar.)

```
;; A thinning records a vertex to be removed and a vector of obs to
;; bypass the outputs of that vertex.  Some thinnings are examples
;; (that is, they refer to vertices in thinning example graphs) and
;; some are "real" (that is, they refer to vertices in graphs in the
;; current grammar).  All real thinnings are derived from some
;; example, and are linked to that example through thinning-exth.

(defstruct (thinning
             (:print-function print-thinning)
             (:include dlist))
  tv            ; vertex to be removed
  bypass        ; vector of obs for bypass
  weight        ; numeric weight, or nil
  exth          ; if example, nil; if real, link to example
  obligations)  ; dlist of thinning obligations, or nil
```

Figure 6.11: The representation for a thinning

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 re-uses 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 there's 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's 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 re-use of previously created nonterminals be important?

Consider the Fibonacci example from Section 5.6. In that example, we pretended that there was only one way to thin the example in Figure 5.5. In fact, there are four, as shown in Figure 6.12 below. 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 5.8. At this point Thinner assigns a weight to each of the four possibilities. Without considering the question of X_β re-use, 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).

In short, the re-use of a previously created nonterminal X_β is desirable because it means the re-use of a *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 2.12). If we always thin 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.

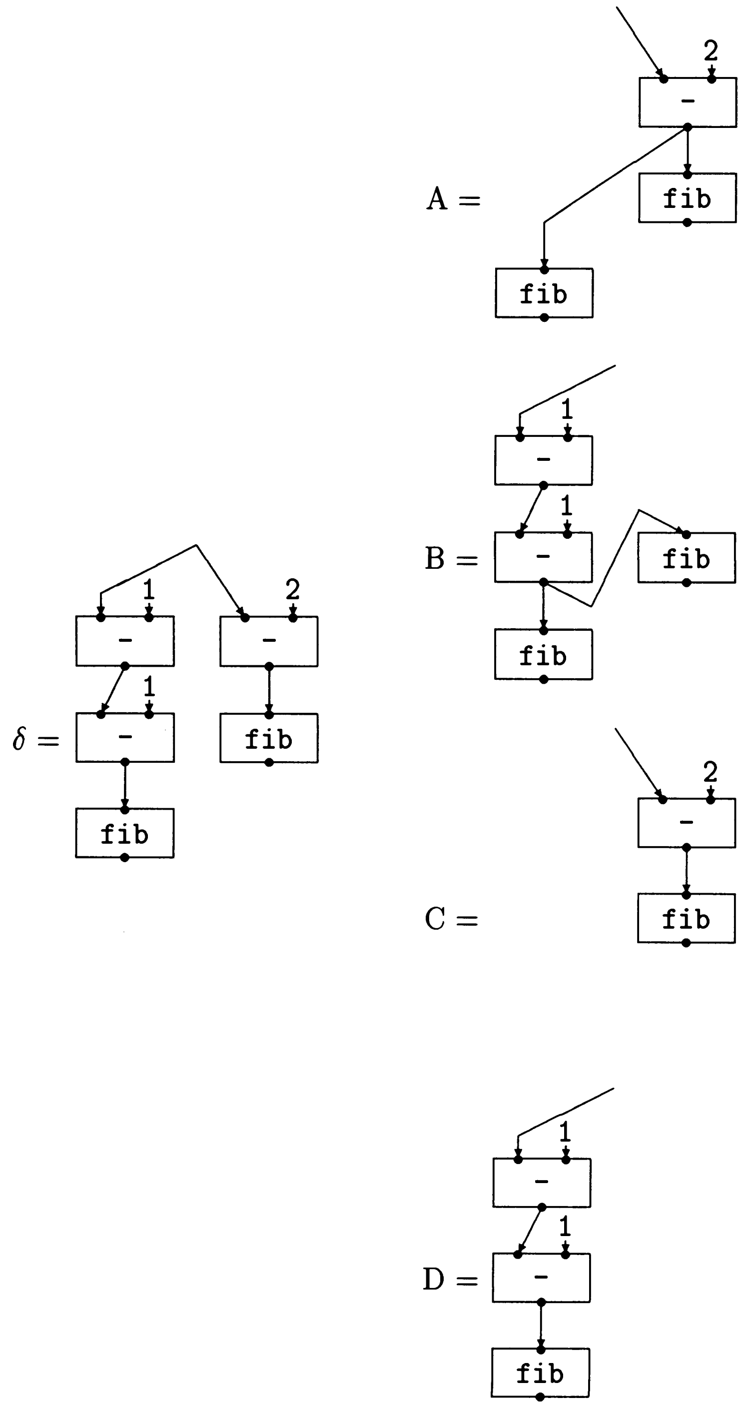


Figure 6.12: Four thinning possibilities

We've already observed that Thinner maintains a *normal* grammar at all times. This means that it won't 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.

Figure 6.13 shows the routine Thinner uses to assign a weight to a thinning. Note that it checks whether any predecessor of a predicate is altered by the thinning, and if so it computes any thinning obligations. The thinning is performed provisionally, weighed, and then retracted. The final weight includes the weight of any thinning obligations.

6.7.3 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're 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've 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.

```
;; (weigh-thinning th ex kmap) sets (thinning-weight th) for a real
;; thinning: one that isn't just an example but really thins some
;; graph in a function.  We test whether the thinning alters the
;; predecessors of any predicates; if so, we compute the thinning
;; obligations -- thinnings of other graphs in the same function --
;; and establish a dlist of them in (thinning-obligations th).  The
;; weight of the thinning reflects the excised vertices; whether the
;; thinning generates any kernels; and the weights of the thinning
;; obligations.

(defun weigh-thinning (th ex kmap)
  (setf (thinning-weight th) nil)
  (when (proper-thinning? th)
    (let* ((v (thinning-tv th))
           (g (vertex-graph v))
           (f (graph-function g))
           (gpreds (get-altd-preds th)))
      (when gpreds (validate-dtrees f g))
      (multiple-value-bind (cutv b dead) (do-thinning th)
        (when gpreds (get-obligations ex th gpreds kmap cutv dead))
        (when (or (not gpreds) (thinning-obligations th))
          (set-cut-weight th cutv)
          (when (thinning-obligations th)
            (for oth :indlist (thinning-obligations th)
              :do (incf (thinning-weight th)
                        (thinning-weight oth))))
          (add-overlap-weight th kmap))
        (abort-thinning cutv b dead))
      (when gpreds (free-vector gpreds))))
  nil)
```

Figure 6.13: Weighing a thinning, with obligations

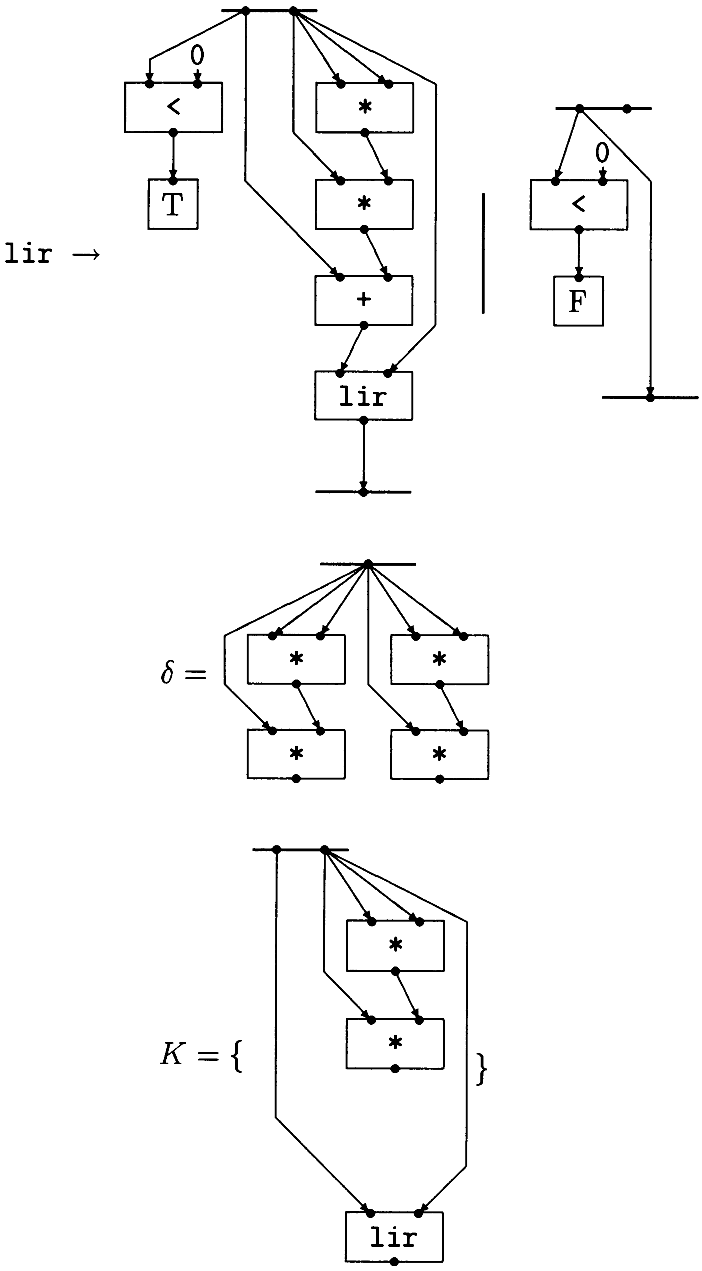


Figure 6.14: A thinning problem that makes grouping difficult

Figure 6.14 shows an initial grammar, a thinning example, and a kernel set. (The program requires loop invariant removal: the two multiplications are loop invariant and should be computed only once. Thinner's performance on this example is discussed further in Section 6.8.4.) Observe that there is an instance of the kernel in the first production for `lir`. Let's match the kernel as a subgraph of the graph and try to form the appropriate group. The first multiplication is in the group; the second multiplication is in the group; its successor, the addition, is *not* in the group; its successor, the nonterminal `lir`, is in the group again. So if we try to replace this group with a new nonterminal in the obvious way, we end up introducing a cycle as shown in Figure 6.15.

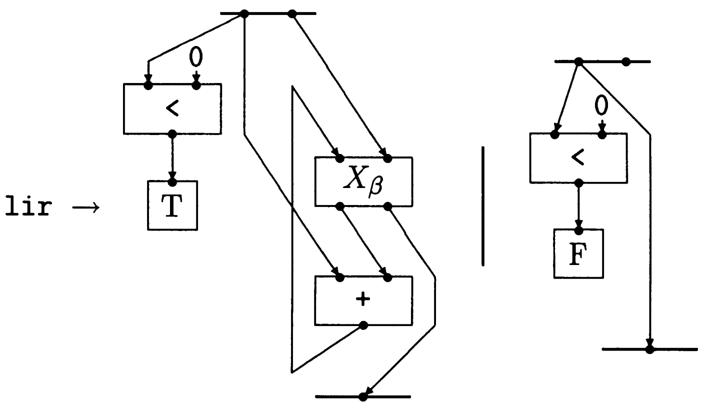


Figure 6.15: Result of naive grouping: a cycle

Any implementation of the method of kernels for trace grammars must deal with this problem. I experimented with two approaches. The first involves allowing group membership to bleed from group vertices to other, neighboring vertices as necessary to prevent cycles. (In the example this means expanding the group to include the addition.)

The second approach involves freezing vertices, eliminating them from the group. It starts by detecting any vertex in the group from which another vertex in the group can be reached by a path that leaves the group. (In the case of the example, the two multiplications pass this test.) These vertices are then frozen: they are not included in the group, but their outputs are recorded as being available for bypasses in the new nonterminal. If they are used when the new nonterminal is thinned, these values end up as new function inputs.

Neither approach is satisfactory in all cases. Bleeding is easier and is always possible, but it is often the wrong thing to do. The re-use of nonterminals, the importance of which has already been noted, is curtailed by spurious additions to groups. Freezing is safer but is not always an option, since vertices that are going to be thinned cannot be removed from the group.

Thinner uses freezing whenever it can. (In the loop invariant example, it freezes the two multiplications.) Occasionally it encounters a group that can't be decircularized by freezing alone, and then it resorts to bleeding. This combination suffices for many examples, but I suspect that a more general solution to the problem may be possible. Perhaps one could go ahead and introduce circularities, carry though the method of kernels, and then remove any circularities in a final step.

6.7.4 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.

I 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, I 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 basic kernel set instead: the set of minimal precursors of the thinning example along the path of derivation on which it was discovered. In those cases where this basic set does not include all useful kernels from the exhaustive set, Thinner will miss some thinnings. This occurs in one instance in the transcripts that follow, but the thinning that is missed because of the nonexhaustive kernel set is found again in a later thinning step. (This later step is a thinning with the same example, found on a different path of derivation.)

6.7.5 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 explorer by contracting the search space. The explorer performs fewer unfoldings when fabricated functions have been inlined. This can change the order in which Thinner treats future thinning examples.

6.8 Thinner transcripts

These examples are in the form of transcripts of Thinner sessions. They are exact transcripts, except that some of the variable names generated by Thinner have been changed to improve readability. (Thinner tries to preserve the original variable names, but when it needs new variables it uses `gensym` to make new names.)

The examples use three top-level Thinner functions:

`(suite filename)` Compile the file named by the string `filename` (from Thinner's test-suite directory) and use the resulting trace grammar as the current grammar. This also resets the cumulative timer mentioned below.

`(thin)` Find one violation of the PLC and repair it in the current grammar. (If no violation can be found this function diverges.) When it returns, this function prints the message "Accumulated optimization time:" followed by a measure, in seconds, of the time spent in `thin`; this time is computed using the Common Lisp function `get-internal-run-time`. The timer is cumulative, so it will grow with each call to `(thin)`, until reset by `(suite filename)`.

`(thin n)` The `thin` function takes an optional parameter, which is a count of iterations: `(thin 2)` is the same as two evaluations of `(thin)`.

`(decompile-all)` Decompile the current grammar and print the resulting program.

6.8.1 Common subexpression elimination

Common subexpression elimination is just a finger exercise for Thinner. Common subexpressions could usually be detected without using the method of constraints, just by inspecting the normal forms of expressions, and the optimization could usually be performed just by thinning the right-hand sides of productions in the grammar, without employing the method of kernels. But Thinner does not take any shortcuts. It's a pretty large hammer for little nails like this:

```
> (suite "cse1")
Compiling test program CSE1 (NIL).

(DEFUN CSE1 (A B)
  (CONS (OR A B) (CONS (OR A B) (CONS (OR A B) BNIL))))

NIL
> (thin)
Accumulated optimization time: 0.05s.
NIL
> (decompile-all)

(DEFUN CSE1 (A B)
  (DECLARE (BOOLEAN A) (BOOLEAN B))
  (LET ((C (OR A B))) (CONS C (CONS C (CONS C BNIL)))))
NIL
>
```

Thinner performs *interprocedural* common subexpression elimination with equal facility. The first thinning it finds in this example is a normal common subexpression elimination:

```
> (suite "cse3")
Compiling test program F (NIL).

(DEFUN F (X Y Z) (+ (* Y X) Z))

Compiling test program CSE3 (NIL).

(DEFUN CSE3 (A B C) (F (+ A B) C (* C (+ A B))))

NIL
> (thin)
Accumulated optimization time: 0.04s.
NIL
```

```
> (decompile-all)

(DEFUN CSE3 (A B C)
  (DECLARE (INTEGER A) (INTEGER B) (INTEGER C))
  (LET ((D (+ A B))) (F D C (* C D))))
(DEFUN F (X Y Z)
  (DECLARE (INTEGER X) (INTEGER Y) (INTEGER Z))
  (+ (* Y X) Z))
NIL
```

But ask it for another thinning, and it will find and eliminate the interprocedural common subexpression:

```
   > (thin)
   Accumulated optimization time: 0.18s.
   NIL
   > (decompile-all)

   (DEFUN CSE3 (A B C)
     (DECLARE (INTEGER A) (INTEGER B) (INTEGER C))
     (LET ((D (* C (+ A B)))) (+ D D)))
   (DEFUN F (X Y Z)
     (DECLARE (INTEGER X) (INTEGER Y) (INTEGER Z))
     (+ (* Y X) Z))
NIL
>
```

Note that in this case Thinner has cleaned up a fabricated function. The method of kernels yields this, before cleanup:

```
   (DEFUN CSE3 (A B C)
     (DECLARE (INTEGER A) (INTEGER B) (INTEGER C))
     (G A B C))
   (DEFUN F (X Y Z)
     (DECLARE (INTEGER X) (INTEGER Y) (INTEGER Z))
     (+ (* Y X) Z))
   (DEFUN G (A B C)
     (DECLARE (INTEGER A) (INTEGER B) (INTEGER C))
     (LET ((D (* C (+ A B))))
       (+ D D)))
NIL
>
```

Thinner then unfolds and eliminates the fabricated function G, as discussed above. In the examples below I will let this cleanup pass without further comment.

6.8.2 Dead code elimination and constant folding

Dead code elimination is another easy case of thinning. Here's an example in which one arm of a conditional can never be executed. In this example Thinner reasons that an integer times its successor can never be equal to 1.

```
> (suite "dce1")
Compiling test program DCE1 (NIL).

(DEFUN DCE1 (A) (IF (EQUAL (* A (+ A 1)) 1) A 0))

NIL
> (thin)
Accumulated optimization time: 0.03s.
NIL
> (decompile-all)

(DEFUN DCE1 (A) (DECLARE (INTEGER A)) (IF FALSE A 0))
NIL
> (thin)
Accumulated optimization time: 0.04s.
NIL
> (decompile-all)

(DEFUN DCE1 (A) (DECLARE (INTEGER A)) 0)
NIL
>
```

It actually takes two steps to complete the dead code elimination. The first step notes that the value on which the program branches is constant, while the second thins away the predicate (and one of the two graphs in the executable set of productions for DCE1).

An expression can also be "dead" in the sense that it computes a value which is never used. In the following example, the expression `(= C D)` can never be true; the inference engine discovers this as described in Section 4.2.4. Thinner first eliminates `(EQUAL C D)`, replacing it with the constant false. This lets it eliminate the dead computations of `C` and `D`. Then, in another thinning, it discards the predicate and the graph with no executions.

```
> (suite "dce2")
Compiling test program DCE2 (NIL).

(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)))))

NIL
> (thin)
Accumulated optimization time: 0.06s.
NIL
> (decompile-all)

(DEFUN DCE2 (X Y)
  (DECLARE ((LISTOF INTEGER) X) ((LISTOF INTEGER) Y))
  (MULTIPLE-VALUE-BIND (V0 V1)
      (IF FALSE (VALUES 1 X) (VALUES 0 INIL))
    (LENGTH (CONS V0 V1))))
NIL
> (thin)
Accumulated optimization time: 0.06s.
NIL
> (decompile-all)

(DEFUN DCE2 (X Y)
  (DECLARE ((LISTOF INTEGER) X) ((LISTOF INTEGER) Y))
  (LENGTH (CONS 0 INIL)))
NIL
```

The program is not yet as thin as possible, since the expression (LENGTH (CONS 0 INIL)) is constant. In one more step Thinner discovers this constant folding:

```
> (thin)
Accumulated optimization time: 0.08s.
NIL
> (decompile-all)

(DEFUN DCE2 (X Y)
  (DECLARE ((LISTOF INTEGER) X) ((LISTOF INTEGER) Y))
  1)
NIL
>
```

6.8.3 Code sinking

We've already noted that code sinking in trace grammars works just like dead code elimination. The decision about exactly where to place a conditional is made when the grammar is decompiled.

```
> (suite "sink")
Compiling test program SINK (NIL).

(DEFUN SINK (A B)
  (LET ((X (* A A))) (IF (> A B) (* X B) (* A B))))

NIL
> (thin)
Accumulated optimization time: 0.01s.
NIL
> (decompile-all)

(DEFUN SINK (A B)
  (DECLARE (INTEGER A) (INTEGER B))
  (* (IF (> A B) (* A A) A) B))
NIL
>
```

6.8.4 Loop invariant removal

Here is the example we used in Section 6.7.3 to explain one of the difficulties that occurs when applying the method of kernels to trace grammars—the problem of replacing a group of vertices with a single nonterminal, without introducing cycles.

```
> (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
>
```

Note that Thinner does not know that the function it's optimizing is a "loop". This holds for the other loop optimizations that follow. 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.

6.8.5 Loop splitting

To Thinner, loop invariant removal and loop splitting are quite similar.

```
> (suite "split")
Compiling test program F (NIL).
(DEFUN F (A B C)
  (IF (EQUAL A B)
      C
      (+ (IF (< B C) A (* A A))
         (F (- A 1) B C))))
NIL
```

Thinner observes that the test `(< B C)` is invariant moves it outside of the loop, just as it does with any loop-invariant computation. Then in two more thinning steps, it makes two specialized loop bodies.

```
> (thin 3)
Accumulated optimization time: 1.9s.
NIL
> (decompile-all)

(DEFUN F (A B C)
  (DECLARE (INTEGER A) (INTEGER B) (INTEGER 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)
  (DECLARE (INTEGER B) (INTEGER C) (INTEGER A))
  (IF (EQUAL A B) C (+ A (F1 B C (- A 1)))))
(DEFUN F2 (B C A)
  (DECLARE (INTEGER B) (INTEGER C) (INTEGER A))
  (IF (EQUAL A B) C (+ (* A A) (F2 B C (- A 1)))))
>
```

6.8.6 Loop jamming

Loop jamming is another traditional compiler optimization Thinner can perform. The following function `F` takes integer arguments `I` and `START` and returns two values: the sum of `START` and the integers from 0 to `I`, and the sum `START` and the squares of the integers from 0 to `I`. These two values are computed in separate loops, and naturally we want to jam the loops together. Thinner accomplishes this optimization, although by the fortunes of blind search it finds several others first.

```
> (suite "jam")
Compiling test program SUM (NIL).

(DEFUN SUM (I SOFAR)
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))

Compiling test program SUMSQ (NIL).

(DEFUN SUMSQ (I SOFAR)
  (IF (EQUAL I 0) SOFAR (SUMSQ (- I 1) (+ SOFAR (* I I)))))

Compiling test program F (NIL).

(DEFUN F (I START) (VALUES (SUM I START) (SUMSQ I START)))

NIL
```

Thinner first observes that near the end of the recursion in `SUMSQ`, the program will compute `(* I I)` when `I` is 1.

```
> (thin)
Accumulated optimization time: 0.58s.
NIL
```

```
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER START))
  (VALUES (SUM I START) (SUMSQ I START)))
(DEFUN SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
NIL
```

Next, Thinner observes that, when both `SUM` and `SUMSQ` are unfolded in the body of `F`, there are two computations of `(= I 0)`. This looks like the loop jamming we were hoping for. It takes several thinning steps since there is dead code to be eliminated: when the two instances of `(= I 0)` are made into one, Thinner realizes that one cannot be true and the other false, and eliminates the unreachable code. This captures the idea that `SUM` and `SUMSQ` must terminate after the same number of iterations.

```
> (thin 4)
Accumulated optimization time: 2.01s.
NIL
```

```
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER START))
  (IF (EQUAL I 0)
      (VALUES START START)
      (LET ((X (- I 1)))
        (VALUES (SUM X (+ START I)) (G I START X)))))
(DEFUN SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
NIL
```

It seemed like that last step was going to give us loop jamming, but it didn't. It only jammed the first iteration. What went wrong?

The problem lies in Thinner's use of a nonexhaustive kernel set. It got its thinning example by unfolding `SUM` and `SUMSQ` in the body of `F`, and this path of derivation was used to construct the basic kernel set. But there is another way to get the same thinning example: by unfolding `SUM` and the fabricated function `G` together. If Thinner had constructed an exhaustive kernel set, it would have found this and completed the loop jamming. (Or if it had found this example before constructing the function `G`, the basic kernel set would have been sufficient.)

But as things stand, another instance of the same thinning example is still there to be found. It will only be found when Thinner unfolds `SUM` and `G` in the body of `F`—the explorer will do this when it looks for examples at depth two, but it now has additional examples at depth one. In particular, it identifies a redundant computation of the value (+ START I). When G is unfolded in the body of F it computes that value twice.

```
> (thin)
Accumulated optimization time: 2.51s.
NIL
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER START))
  (IF (EQUAL I 0)
      (VALUES START START)
      (LET ((M (- I 1)))
        (MULTIPLE-VALUE-BIND (V0 V1)
            (IF (EQUAL M 0)
                (LET ((N (+ START I))) (VALUES N N))
                (VALUES (G M (+ START (* I I)) (- M 1))
                        (+ START I)))
          (VALUES (SUM M V1) V0)))))
(DEFUN SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
NIL
```

Next, still at depth one, there is a redundant computation of (EQUAL M 0). When SUM is unfolded in the body of F it computes (EQUAL M 0) twice.

```
> (thin 2)
Accumulated optimization time: 3.26s.
NIL
```

```
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER START))
  (IF (EQUAL I 0)
      (VALUES START START)
      (LET ((M (- I 1)))
        (IF (EQUAL M 0)
            (LET ((N (+ START I))) (VALUES N N))
            (VALUES (SUM (- M 1) (+ (+ START I) M))
                    (G M (+ START (* I I)) (- M 1)))))))
(DEFUN SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
NIL
```

Now there is an optimization at depth zero: a redundant computation of `(- M 1)` in the body of `F`. (Note that the computation of `(+ START I)` is *not* redundant.)

```
> (thin)
Accumulated optimization time: 3.45s.
NIL
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER START))
  (IF (EQUAL I 0)
      (VALUES START START)
      (LET ((M (- I 1)))
        (IF (EQUAL M 0)
            (LET ((N (+ START I))) (VALUES N N))
            (LET ((P (- M 1)))
              (VALUES (SUM P (+ (+ START I) M))
                      (G M (+ START (* I I)) P)))))))
```

```
(DEFUN SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
NIL
```

Finally, there are no more thinning examples to be found before depth two. Thinner now finds the other way to generate the thinning example for loop jamming, the one we missed before by using a nonexhaustive kernel set. Unfolding both SUM and G in the body of F yields a graph that computes (EQUAL P 0) twice. Subsequent thinnings clean up the dead code that results, and this completes the optimization.

```
> (thin 4)
Accumulated optimization time: 8.1s.
NIL
> (decompile-all)

(DEFUN F (I START)
  (DECLARE (INTEGER I) (INTEGER 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 SUMSQ (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (G I SOFAR (- I 1))))
(DEFUN SUM (I SOFAR)
  (DECLARE (INTEGER I) (INTEGER SOFAR))
  (IF (EQUAL I 0) SOFAR (SUM (- I 1) (+ SOFAR I))))
```

```
(DEFUN G (I SOFAR Y)
  (DECLARE (INTEGER I) (INTEGER SOFAR) (INTEGER Y))
  (IF (EQUAL Y 0)
      (+ SOFAR 1)
      (G Y (+ SOFAR (* I I)) (- Y 1))))
(DEFUN H (A B C D)
  (DECLARE (INTEGER A) (INTEGER B) (INTEGER C) (INTEGER D))
  (IF (EQUAL D 0)
      (VALUES B (+ A 1))
      (H (+ A (* C C)) (+ B D) D (- D 1))))
NIL
>
```

6.8.7 Fibonacci reformulation

This is the Fibonacci example we walked through in Section 5.6. The first thinning step does most of the work, making the running time linear.

```
> (suite "fib")
Compiling test program FIB (NIL).

(DEFUN FIB (A) (IF (< A 2) A (+ (FIB (- A 1)) (FIB (- A 2)))))

NIL
> (thin)
Accumulated optimization time: 0.28s.
NIL
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2)
      A
      (MULTIPLE-VALUE-BIND (X Y) (F A (- A 1)) (+ Y X))))
```

```
(DEFUN F (A B)
  (DECLARE (INTEGER A) (INTEGER B))
  (IF (< B 2)
      (VALUES (FIB (- A 2)) B)
      (MULTIPLE-VALUE-BIND (X Y)
          (F B (- B 1))
        (LET ((T4069 (- A 2))) (VALUES Y (+ Y X))))))
NIL
```

This is good, but not yet as thin as possible. There is an unnecessary computation of `(- A 2)`. A second effort removes it:

```
> (thin)
Accumulated optimization time: 0.33s.
NIL
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2)
      A
      (MULTIPLE-VALUE-BIND (X Y) (F A (- A 1)) (+ Y X))))
(DEFUN F (A B)
  (DECLARE (INTEGER A) (INTEGER B))
  (IF (< B 2)
      (VALUES (FIB (- A 2)) B)
      (MULTIPLE-VALUE-BIND (X Y)
          (F B (- B 1))
        (VALUES Y (+ Y X)))))
NIL
```

We're still not done. Thinner notes that in `F`, if `(< B 2)`, then `A` must be 2.

```
> (thin)
Accumulated optimization time: 0.68s.
NIL
```

```
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2)
      A
      (MULTIPLE-VALUE-BIND (X Y) (F (- A 1)) (+ X Y))))
(DEFUN F (B)
  (DECLARE (INTEGER B))
  (IF (< B 2)
      (VALUES B (FIB 0))
      (MULTIPLE-VALUE-BIND (X Y)
          (F (- B 1))
        (VALUES (+ X Y) X))))
NIL
```

Now `(FIB 0)` is a constant, which Thinner eliminates in two more steps:

```
> (thin 2)
Accumulated optimization time: 1.21s.
NIL
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2)
      A
      (MULTIPLE-VALUE-BIND (X Y) (F (- A 1)) (+ X Y))))
(DEFUN F (B)
  (DECLARE (INTEGER B))
  (IF (< B 2)
      (VALUES B 0)
      (MULTIPLE-VALUE-BIND (X Y)
          (F (- B 1))
        (VALUES (+ X Y) X))))
NIL
```

We're still not done. For `(FIB 2)` the program will compute `(+ X Y)`, where `Y` is zero. Thinner repairs this in the next step:

```
> (thin)
Accumulated optimization time: 1.5s.
NIL
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2) A (MULTIPLE-VALUE-BIND (X Y) (H (- A 1)) X)))
(DEFUN H (B)
  (DECLARE (INTEGER B))
  (IF (< B 2)
      (VALUES B B)
      (MULTIPLE-VALUE-BIND (X Y)
          (H (- B 1))
        (VALUES (+ X Y) X))))
NIL
```

In fact, Thinner never stops finding optimizations for this function. For each subsequent thinning, it extracts the next base case of the computation and renders it as a constant. If we thin, say, five more times we'll end up with this:

```
> (thin 5)
Accumulated optimization time: 5.21s.
NIL
> (decompile-all)

(DEFUN FIB (A)
  (DECLARE (INTEGER A))
  (IF (< A 2)
      A
      (LET ((B (- A 1)))
        (IF (< B 2)
            B
            (LET ((C (- B 1)))
              (IF (< C 2)
                  2
                  (LET ((D (- C 1)))
                    (IF (< D 2)
                        3
                        (LET ((E (- D 1)))
```

```
             (IF (< E 2)
               5
               (LET ((F (- E 1)))
                (IF (< F 2)
                  8
                  (MULTIPLE-VALUE-BIND (X Y)
                    (H (- F 1))
                   X)))))))))))))
(DEFUN H (B)
 (DECLARE (INTEGER B))
 (IF (< B 2)
   (VALUES 13 8)
   (MULTIPLE-VALUE-BIND (X Y)
     (H (- B 1))
    (VALUES (+ X Y) X))))
NIL
>
```

6.9 What Thinner can't do

First of all, Thinner can't 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.

Because it uses the relational constraint method of inference, Thinner will sometimes miss thinning examples that seem reasonably clear to a human programmer. For example, consider this function:

```
(defun lir2 (a b c)
    (declare ((listof integer) c))
    (if (or (= a inil) (> (length c) (length a)))
        b
        (lir2 (cdr a) (cons (car a) b) (cdr c))))
```

The expression `(> (length c) (length a))` is loop-invariant, since lists `a` and `c` shrink in unison; but Thinner does not detect this.

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 violations 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. As the Fibonacci example shows, Thinner may start materializing the constant bases cases of a recursion. 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.

Chapter 7

Conclusion

7.1 Summary

In the first chapter I began by noting that in terms of power and flexibility, people make better optimizers than machines. Observing that people make use of high-level principles of optimization, I set out to develop an automatic optimizer with this same ability: the ability to derive useful optimizations as necessary from a high-level, intuitive principle.

The remaining chapters fought for this goal. The battle for an ambitious, underspecified goal like this is always a rear-guard action: the trick is to retreat into a territory that is implementable while yielding as little ground as possible. But some ground is inevitably lost. Let's review the results:

- I decided to stake out a single principle (the Principle of Least Computation) and a single domain (the domain of TL programs). Human programmers don't operate with such a restriction, so this certainly represents a significant reduction in territory. For a single dissertation such a restriction is inevitable.

- In Chapter 2 I chose a formalization of the principle. This formal statement of the PLC (Definition 2.21) in terms of the thinner-than relation on trace grammars may agree with the informal statement, but there can be no proof of this. I acknowledge that I may have given up more ground here.

- The question of whether a thinning is conservative or not is undecidable (Theorem 2.4). It follows that ground must be lost here too: we cannot identify all opportunities for thinning. But this undecidability is less important in practice than the intractability of general inference.[1] To deal with this intractability I developed in Chapter 4 the method of relational constraint, which completes in polynomial time but, naturally, gives up more territory.

- In Chapter 5 I chose a method for repairing violations of the principle. I found an approximate solution to the grammar-thinning problem for context-free grammars. This solution, the method of kernels, applies reasonably well to trace grammars. But the method is not perfect. One can construct a grammar and a thinning example for which the method yields an optimized grammar that still makes the mistake illustrated in the example.

- A final loss of ground occurred in the implementation of Thinner. I chose to use a non-exhaustive kernel set and a simple method of searching for thinning examples.

Thinner itself delineates the territory that remains. It discovers a large, uniform space of optimizations, and demonstrates that the ideas developed in this thesis

[1] In my earliest sketch of a design for Thinner I drew a box labeled "Inference Engine". I have learned to see this as a serious mistake. One might as well draw a box labeled "Lots of Computation" or "A Miracle Occurs". Thinner would not have been successful without efficient inference, and general-purpose *efficient* inference is not available as an off-the-shelf component.

are practical. It is possible to derive optimizations as needed from an intuitive, high-level principle.

7.2 Open problems

We have seen that the path from the idea of principled optimization to its realization in Thinner involved many compromises. Each of these compromises represents an open problem: can it be improved?

- There are certainly other principles of optimization and other interesting domains of computational mechanisms. I think the domain of sequential digital circuits would be an interesting one to investigate. The PLC applies here, but less absolutely: sometimes a little redundancy yields a better layout and hence a better circuit. As in real-world planning problems, there are resource constraints to consider.

  The digital circuit domain is an example of a domain for which there is no well-established canon of rote optimizations. Applying compiler optimizations to the source code used in high-level circuit synthesis sometimes makes the circuit worse; this is noted by McFarland, Parker and Camposano as one of the important open problems in circuit synthesis [MPC88]. Principled optimization may provide some leverage on this problem. (It is interesting to note in this context that the high-level circuit synthesis community has recently shown increased interest in functional languages [JWB89] [BT89] [Joh83].)

- The formalization of the PLC could be expanded. The most significant weakness of the current formalization is that it requires substitution with identity, where substitution with adequacy might be more natural. For example, if a program computes `(length a)`, and if there is a list `b` available which, though not equal to `a`, has the same length as `a`, then we might want to be able to substitute `b` for `a` and eliminate the computations that generated `a`. This transformation is not justified by my formalization of the PLC since it is not a conservative thinning. But it may well be justified by the intuitive principle, so perhaps the formalization should be strengthened.

I imagine defining another class of thinnings, the *liberal* thinnings, that allow substitution with adequacy. But the notion of adequacy is tricky. Should we insist that a function remain adequate with respect to some specification? This is the approach taken by Hall [Hal90], but I am uncomfortable with the idea of depending on external specifications since, in practice, functions that come with formal specifications are rare. Should we insist on functional equivalence and allow substitution with adequacy only for intermediate values?

- Greater inference power could be developed. Trace grammars often suggest simple inductions: a nonterminal will often have obvious "base cases" (fully terminal productions) and "inductive cases" (recursive productions). Perhaps a method for induction could be added to Thinner's semantic analyzer.

- The CFG thinning problem is open. Any solution to the CFG thinning problem may yield an optimization technique for trace grammars, as the method of kernels did. This potential application makes the CFG thinning problem especially interesting.

  One CFG thinning solution is the FSM method (this is the one that works on fully terminal thinning examples δ with $M(\delta)$ regular, from Section 5.2.1).

I have not yet examined this to see whether it might yield an optimization technique for trace grammars. If it does, the restriction that it only applies when δ is fully terminal might be relaxed: in the CFG case one can sometimes treat nonterminals in δ as if they were terminal (provided such nonterminals do not contribute to instances of δ after expansion).

- The implementation limitations of Thinner might yield to further research. In this area two advances would be especially useful: a way to construct a larger collection of useful kernels at reasonable expense, and a more intelligent way to search the space of possible thinning examples.

7.3 The benefits of principled optimization

The problems addressed in this thesis have not yet received wide attention, so I will conclude by explaining why I think they are worth studying.

7.3.1 The science of optimization

Flip through any standard text on compiler construction and you will find a collection of recommended compiler optimizations. Decades of research and development and thousands of person-years of field work have produced this treasure, the modern canon of compiler optimizations. Let any who would alter this canon beware: with so much research behind it, how can it be far from optimal?

While this canon is of unquestionable value to compiler engineers, it leaves many questions unanswered. From compiler texts one gets the impression that the canon is everything: optimization is just a list of special cases, a grab-bag of hacker's tricks. But where's the science in that?

Thinner demonstrates that a variety of compiler optimizations that were formerly thought of as independent tricks are actually instances of a single principle at work. It unifies and illuminates these well-known optimizations by showing that they can be derived automatically from the PLC. There *is* a science of optimization, and it is found in the study of the principles that justify optimization.

Looking to the future: if it took decades of research and thousands of person-years to come up with this canon for Von Neumann architectures, must we repeat that effort to develop satisfactory compilers for each new architecture? Perhaps, with advances in the science of optimization, we will be able to develop optimizations by some more refined method than trial and error.

7.3.2 The tools of optimization

The development of Thinner has produced two tools which may be useful outside of principled optimization: the relational constraint method of inference, and the method of kernels for program transformation.

The relational constraint method is a fast method of inference which should be applicable to other areas. Whenever *part of* an inference task can be treated as constraining k-ary relations for fixed k, and whenever *part of* that vocabulary of relations can be captured in a finite lattice, the relational constraint method may be of assistance. We have already observed that the method would be easy to parallelize.

The method of kernels is not restricted to thinning, but could apply to a variety of different optimizations. It serves to expose instances of a particular example graph in the right-hand sides of productions in the grammar. From there, what you do to transform the graph is relatively unconstrained. Nor is the method of kernels the last word on grammar thinning. The more important, fundamental contribution is the observation that one can exploit the connection between program optimization and grammar transformation.

7.3.3 Practical applications

Thinner is a proof of concept, not an application; there are no TL programs outside the Thinner project. But I believe that principled optimization will ultimately find a place in practical optimizing tools.

Let me make it clear that I am not advocating the construction of a practical optimizer that painfully rediscovers loop invariant removal over and over. The world does not need another extremely slow compiler. The canon of rote compiler optimizations has its place: these are precisely the optimizations that experience shows are easy to find, and occur frequently enough to be worth looking for. The canon is both necessary and, in most cases, sufficient. Studies suggest that state-of-the-art optimizing compilers find most of the optimization there is to be found in most programs, so that further optimization by hand rarely yields as much as a factor of 2 in speed [Pot91].

Expert human programmers rarely write programs that contain egregious violations of the PLC. It is, after all, an intuitive principle of optimization with which we are extremely familiar. I believe that principled optimization has little practical potential for programs *written by human experts.*

But the future of software engineering holds more and more mechanical assistance for programmers: more and more programs *written by machines*, or written by novices with mechanical assistance. I do not refer to full-blown automatic programming only, but to the trend toward higher-level tools and higher levels of abstraction in general: object libraries, very high level languages, specification languages, proofs as programs, etc. We already see that many compiler optimizations are useful largely because they repair inefficiencies introduced by high-level language compilation: most loop invariants, for example, are array indexing instructions not explicit in the source text.

The well-known "argument from re-use" draws on this idea. This is the argument that high-level optimization techniques will promote the re-use of code from procedure and object libraries, by allowing general routines to be specialized automatically [Pot91] [Hal90] [Tur86]. In effect, this argument says that while the original, human-coded functions may not benefit from advanced optimization, machine-generated specializations of them will.

I believe that as we move to high-level tools and higher levels of abstraction, we must be prepared to build in greater optimizing sophistication. The expert human programmer's actions are informed by principles of optimization, and if lower-level actions are to be carried out by tools, these tools must have access to similar principles. This is the practical importance of the study of the principles of optimization.

Epilogue

Burn This Thesis

Seized with a prolific, vernal mood
It sings again the old arboreal hymn:
A verse, a year of life, a layer of wood,
An ever tighter trap of trunk and limb.
I weave, my hands are knotted on the loom,
I think, my mind divided into thought,
I feel a joyful urge to build a tomb,
I act, the walls solidify, I'm caught.
And God is everywhere. What must it mean
To make the world and with the world to join!
How must He brood within each dark machine
As He receives Himself in each dark coin!
Come, devour, plane, exalt, make low,
Release us in a gentle even heat.

Bibliography

[AH87] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1. Ellis Horwood Limited, 1987.

[AN87] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *PARLE: Parallel Architectures and Languages Europe*, volume 2. Springer-Verlag, June 1987.

[Bac85] John Backus. From function level semantics to program transformation and optimization. In H. Ehrig et al., editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 60–91. Springer-Verlag, March 1985. Lecture Notes in Computer Science 185.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1), January 1977.

[BH90] H. Bunke and B. Haller. A parser for context free plex grammars. In M. Nagl, editor, *WG '89 Graph-Theoretic Concepts in Computer Science: Proceedings of the 15th International Workshop WG '89.* Springer-Verlag, 1990. Lecture Notes in Computer Science 411.

[Bib91] W. Bibel. Perspectives on automated deduction. In Robert S. Boyer, editor, *Automated Reasoning*, chapter 4. Kluwer Academic Publishers, 1991.

[BM75] Robert S. Boyer and J. Strother Moore. Proving theorems about LISP functions. *Journal of the Association for Computing Machinery*, 22(1):129–144, January 1975.

[BT89] Alexandre Bronstein and Carolyn L. Talcott. Formal verification of pipelines based on string-functional semantics. In Luc J. M. Claesen, editor, *International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 347–364. Elsevier, November 1989.

[C+86] Robert L. Constable et al. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501. ACM, January 1993.

[CH78] Patrick Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, 1978.

[Cho62] Noam Chomsky. Context-free grammars and pushdown storage. In *Quarterly Progress Report No. 65*, pages 187–194. MIT Res. Lab. Elect., 1962.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[CS70] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers; Preliminary Notes.* Courant Institute of Mathematical Sciences, New York University, 1970.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, October 1980.

[Ers78] A. P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts.* North-Holland, 1978.

[Ers82] A. P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18, 1982.

[Fed71] Jerome Feder. Plex languages. *Information Sciences*, 3:225–241, 1971.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[FLS64] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. *The Feynman Lectures on Physics*, volume 2. Addison-Wesley, 1964.

[Fut71] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5), 1971.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30, 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91 Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 169–192. Springer-Verlag, April 1991. Lecture Notes in Computer Science 493.

[GS90] C. A. Gunter and D. S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 12. The MIT Press/Elsevier, 1990.

[Hal90] Robert Joseph Hall. Program improvement by automatic redistribution of intermediate results. Technical Report AI-TR 1251, MIT AI Lab, December 1990.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Hud87] Paul Hudak. A semantic model of reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 3. Ellis Horwood Limited, 1987.

[Hug87] John Hughes. Analysing strictness by abstract interpretation of continuations. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4. Ellis Horwood Limited, 1987.

[IK87] Klaus Indermark and Herbert Klaeren. Compiling fibonacci-like recursion. *SIGPLAN Notices*, 22(6):101–107, June 1987.

[JGB+90] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.

[JM84] J.P. Jones and Y.V. Matijasevic. Diophantine representation of the set of prime numbers. *Journal of Symbolic Logic*, 49, 1984.

[Joh83] Steven Dexter Johnson. *Synthesis of Digital Designs from Recursion Equations.* Ph.D. dissertation, Indiana University, May 1983. MIT Press, 1984.

[Joh86] J. Howard Johnson. Rational equivalence relations. *Theoretical Computer Science*, 47:39–60, 1986.

[JWB89] Steven Dexter Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification. In Luc J. M. Claesen, editor, *International Workshop on Applied Formal Methods for Correct VLSI Design.* Elsevier, November 1989.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.

[KS81] R. B. Kieburtz and J. Shultis. Transformations of FP program schemes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* ACM, October 1981.

[LN73] Hans Lausch and Wilfried Nöbauer. *Algebra of Polynomials.* North-Holland, 1973.

[MPC88] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *25th Design Automation Conference*, pages 330–336. ACM/IEEE, 1988.

[MR79] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(2), 1979.

[Myc87] Alan Mycroft. A study on abstract interpretation and 'validating microcode algebraically'. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 9, pages 199–218. Ellis Horwood Limited, 1987.

[Per89] Mark Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 122–128, 1989.

[Pot91] I. V. Pottosin. Analysis of program optimization possibilities and further development. *Theoretical Computer Science*, 90:17–36, 1991.

[Ric86] Charles Rich. A formal representation for plans in the Programmer's Apprentice. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, 1986.

[Roh89] J. S. Rohl. Recursive and iterative functions for generating Fibonacci numbers. Technical Report TR 89-1017, Cornell University, June 1989.

[Roz87] Grzegorz Rozenberg. An introduction to the NLC way of rewriting graphs. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science.* Springer-Verlag, 1987. Lecture Notes in Computer Science 291.

[RW88] Charles Rich and Richard C. Waters. A research overview. *Computer*, 21(11):11–25, November 1988.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computation. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.

[Sal73] Arto Salomaa. *Formal Languages.* Academic Press, 1973.

[SG87] Devika Subramanian and Michael R. Genesereth. The relevance of irrelevance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[Ste90] Guy Steele. *Common Lisp: the Language, 2nd Edition.* Digital Press, 1990.

[Sub89] Devika Subramanian. *A Theory of Justified Reformulations.* Ph.D. dissertation, Stanford University, March 1989.

[TNT82] Valentin F. Turchin, Robert M. Nirenberg, and Dimitri V. Turchin. Experiments with a supercompiler. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 47–55, August 1982.

[Tur80] Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Automata, Languages and Programming, the 7th Colloquium*, pages 645–657. Springer-Verlag, 1980. Lecture Notes in Computer Science 85.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood Limited, 1987.

[Web92a] Adam Brooks Webber. A formal definition of unnecessary computation in functional programs. Technical Report TR 92-1260, Cornell University, January 1992.

[Web92b] Adam Brooks Webber. Relational constraint: a fast semantic analysis technique. Technical Report TR 92-1319, Cornell University, December 1992.

[Web92c] Adam Brooks Webber. Thinning context-free languages. Technical Report TR 92-1318, Cornell University, December 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.