A Formal Definition of Unnecessary Computation in Functional Programs

Adam Brooks Webber*

TR 92-1260
January 1992

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

*Supported by NSF grant IRI - 8902721

A Formal Definition of Unnecessary Computation In Functional Programs

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

December 16, 1991

Abstract

Our goal is to develop a new and highly flexible approach to program optimization. Instead of applying rote, high-level transformations, we seek to derive optimizations automatically from broad and intuitive principles. Toward that end this paper presents a new formalism for first-order, purely functional programs, then uses the formalism to give a rigorous statement of a principle of optimization. The formalism occupies three levels. At the lowest level is the *trace graph*, a finite, graph-like structure that describes a single terminating path of execution through a functional program. At the middle level is the trace graph set, which describes a set of paths of execution; a certain kind of trace graph set, the *executable set*, describes the full set of paths for a single deterministic program. At the highest level is the *trace grammar*, a graph grammar that generates a trace graph set. While trace graph sets may be infinite, trace grammars are finite objects with a natural, subroutine-like recursive structure. We use the formalism to give a rigorous statement of a well-known principle of optimization, namely, that programs should not make any unnecessary computations. This principle is so obvious that it is often overlooked, but it underlies many common compiler optimizations and other, more exotic program transformations. Our formal statement of the principle unifies and illuminates many optimizing transformations. Our work in progress is the construction of an optimizer that derives optimizations directly from our formal principle. This paper concludes with an overview of this optimizer and some preliminary experimental results.

1 Introduction

Flip through any standard text on compiler construction and you will find a real treasure: a collection of recommended compiler optimizations. These few basic transformational rules have been refined over decades from a vast body of experience with optimization. Decades of research and development and thousands of person-years of field work have yielded 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?

*Supported by NSF grant IRI-8902721.

While this canon is of unquestionable value to practical program optimizers, it is not the final word on optimization. Many questions about optimization remain unanswered:

- Is optimization just a list of special cases, mere tricks of the trade, or is there more to it?
- What should an optimizer do when its list of transformations is exhausted but the program still isn't fast enough? Must it give up? If so, why don't human optimizers have to give up at the same point? Must optimizers always be stumped by patterns of program inefficiency not explicitly foreseen by the designer?
- 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? Can't we develop optimizations by some more refined method than trial and error?

We believe that the answer to these questions lies in the study of the principles of optimization. To make a start in this direction we have chosen to study a single programming model and a single principle of optimization.

The programming model is the functional or applicative model, and the principle is, simply, that programs should not make any unnecessary computations. We call this the Principle of Least Computation[1] or PLC. In the domain of purely functional programs there are no side-effects, so the only reason to perform a computation is to acquire its result. It follows that there are two ways for a computation to be unnecessary: the result may already be known, or it may not really be needed.

Definition 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.*

This paper begins with a new formalism for first-order, purely functional programs, then uses this formalism to give a rigorous statement of the PLC. The PLC is so obvious that it is often overlooked, but it underlies many common compiler optimizations and other, more exotic program transformations. After our development of a formal statement of the PLC, we give examples showing that a variety of seemingly unrelated optimizations are actually instances of the PLC at work.

Construction of a PLC-based optimizer is under way at this time. This paper concludes with some preliminary experimental results that indicate that our formulation of the PLC is workable and that our goal of automatically deriving optimizations from it is attainable.

2 Related Work

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

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

The transformational approach 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] Thus, whilst certainly being very generally applicable, the unfold/fold methodology requires precise guidance from the programmer, and its use resembles program design in the conventional sense. Full automation would appear difficult, although it may be possible for various collections of standard rule sequences to be maintained in a 'transformation library' and applied as higher-level transformation tactics. [FH88, page 456]

We believe that formal principles of optimization like the PLC can be used to guide the composition of fine-grained transformations.

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 highly principled 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). We are not concerned with identifying high-level patterns of unnecessary computation; we give, in effect, a low-level definition of "unnecessariness".

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 we believe that they are relatively independent of the algebraic approach.

There has been a general movement in the direction of semantic justification for advanced optimization which is quite natural: advanced optimizers are hungry for information about the programs on which they work, and when purely syntactic sources of information are exhausted they must turn to semantic sources, in spite of the computational difficulties. Thus we have, for example, the abstract interpretation of Cousot and Cousot [CC77], the "driving" technique of used by Turchin's Supercompiler [Tur86, TNT82, Tur80], and the Plan Calculus of Rich and Waters [RW88, Ric86]. Modern partial evaluators [Ers82, Ber90] also depend heavily on semantic sources of information. Our approach, too, requires the kind of detailed information only available through semantically-aware program analysis.

Principled optimization is similar in spirit to 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.

3 A Formalism for Functional Programs

Our formalism occupies three levels. At the lowest level is the *trace graph*, a finite, graph-like structure that describes a single terminating path of execution through a functional program. At the middle level is the trace graph set, which describes a set of paths of execution; a certain kind of trace graph set, the *executable set*, describes the full set of paths for a single deterministic program. At the highest level is the *trace grammar*, a graph grammar that generates a trace graph set. While trace graph sets may be infinite, trace grammars are finite objects with a natural, subroutine-like recursive structure.

At each level we give a formal definition of the PLC. In the domain of purely functional programs there are no side-effects, so the only reason to perform a computation is to acquire its result. It follows that there are two ways for a computation to be unnecessary: the result may be obtainable by other means, being either constant or equal to some other available result; or it may not really be needed, being neither an input to another computation nor an output of the program. Of course, the claim that a formal definition agrees with an informal one is not subject to ready proof or refutation. But we believe that the formal definition of the PLC developed in the following pages agrees well with this general description.

3.1 The Trace Graph 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.

3.1.1 Trace Graphs

Definition 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 its* output vector v_{out}. *The vertex set* V *is partitioned into five disjoint subsets:*

V_i: *a singleton with input arity 0 and output arity* ≥ 1, *standing for the producer of a computation's inputs.*

V_o: *a singleton with input arity* ≥ 1 *and 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. Each vertex in V_C has associated with it a typed deterministic function with appropriate input and output arities. Vertices in V_C represent primitive functional computations.*

The edge set E is an acyclic set of directed edges; each edge is either a pair $(v_{out}(i), w_{in}(j))$, representing a value produced by one vertex and required by another, or a pair $(c, w_{in}(j))$, representing a constant supplied to a vertex input. There is exactly one edge supplying every vertex input. Vertices and edges are considered to be typed: every vertex input, vertex output, and constant belongs to a type, edges must be typed identically at both ends, and edges into predicates must be of type boolean. For vertices in V_C, the type of the function determines the types of the vertex inputs and outputs.

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:

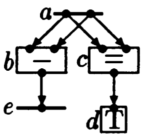
Definition 3 *Vertex v in trace graph G* matches *vertex v' in trace graph G' iff the subgraph of G consisting of v and all its ancestors is isomorphic to the subgraph of G' consisting of v' and all its ancestors.*

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

Figure 1 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. The illustration is not fully specific since doesn't give the type of the functions, but merely suggests that vertex b has the function of subtraction and vertex c the function of comparison. The assumption that they are the usual operations on integers leads to reasonable typing of every vertex in graph: $c_{out}(1)$ and $d_{in}(1)$ get type boolean and all other vertex inputs and outputs get type integer. We will sometimes speak of the type of a graph. This is a functional type with a vector of input types determined by V_i and a vector of output types determined by V_o. For the graph of Figure 1 the type is ((integer, integer) $\rightarrow$ integer).

3.1.2 Trace Graph Executions

A trace graph can be thought of as a partial program that computes a function for a limited set of inputs. An execution of a trace graph given a set of inputs can be defined as follows:



$$
\begin{aligned}
V_i &= \{a\} \\
V_o &= \{e\} \\
V_T &= \{d\} \\
V_F &= \{\} \\
V_C &= \{b, c\} \\
E &= \{(a_{out}(1), b_{in}(1)),\ (a_{out}(1), c_{in}(1)), \\
&\quad (a_{out}(2), b_{in}(2)),\ (a_{out}(2), c_{in}(2)), \\
&\quad (b_{out}(1), e_{in}(1)),\ (c_{out}(1), d_{in}(1))\}
\end{aligned}
$$

Figure 1: Example of a simple trace graph

Definition 4 *Given a trace graph $G = (V, E)$ and an input vector x matching V_i in type, an* execution of G on x *is a function f that assigns a value to each vertex input and output such that:*

- *For the vertex $v \in V_i$ and for all suitable j, $f(v_{out}(j)) = x(j)$.*
- *For each edge $(v_{out}(i), w_{in}(j))$, $f(w_{in}(j)) = f(v_{out}(i))$; for each edge $(c, w_{in}(j))$, $f(w_{in}(j)) = c$.*
- *For all $v \in V_C$, Y is the vector of values assigned by f to v_{in}, Z is the vector of values assigned by f to v_{out}, and g is the function of v iff $g(Y) = Z$.*
- *All vertex inputs in V_T are assigned the boolean value true.*
- *All vertex inputs in V_F are assigned the boolean value false.*

The domain *of a graph,* dom(G), *is the set of input vectors on which G has an execution.*

This is a workable definition even if vertices have partial rather than total functions, which is a situation that we will investigate later. Figure 2 shows an execution of the trace graph from Figure 1 on the input vector $(2, 2)$. (This graph has an execution on an input (n, d) of the correct type if and only if $n = d$.)

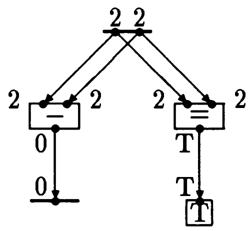


Figure 2: Execution of a trace graph on (2,2)

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.

Theorem 1 *There is at most one execution of a 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 $v_{out}(i)$. Since the graph is acyclic, $v_{out}(i)$ 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 $v_{out}(i)$ is at the root of a tree whose leaves are assigned fixed values and whose internal nodes are deterministic functions, f must assign a fixed value to $v_{out}(i)$. Now take any vertex input $w_{in}(j)$. If it has an edge from a constant, f must assign it that constant; if it has an edge from a vertex output $v_{out}(i)$, f must assign it the same value as $v_{out}(i)$, which is fixed for fixed input vector x. Thus, f is determined by x over its entire range. If x determines the right boolean values for V_F and V_T then there is one execution f of G on x; otherwise, there is no execution of G on x. $\square$

3.1.3 Thinning Trace Graphs

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

Definition 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 and consistently typed.

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

Definition 6 *Let $G = (V, E)$ and $G' = (V', E')$ be trace graphs. $G' \sqsubseteq G$ if and only if there is exist some vertex set $W \subseteq (V_C \cup V_T \cup V_F)$ and some bypass set F such that $((V \setminus W), (E \setminus E_W \cup F))$ is isomorphic to G'. $G' \sqsubset G$ if and only if $G' \sqsubseteq G$ and G' is not isomorphic to G.*

To identify the W and F involved in the thinning, we will sometimes write $G' \sqsubseteq_F^W G$.

Figure 3 shows an example of a thinning. The following theorem summarizes several properties of the $\sqsubseteq$ relation.

Theorem 2 *The following are properties of the $\sqsubseteq$ relation on trace graphs:*

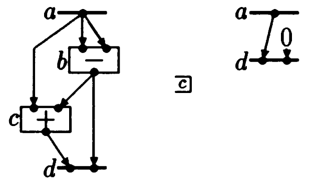


Figure 3: A conservative thinning: $W = \{b, c\}$ and $F = \{(a_{out}(1), d_{in}(1)), (0, d_{in}(2))\}$

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 $G \sqsubseteq_F^W H$ and $G' \sqsubseteq_{F'}^{W'} H$ then there is a graph G'' such that $G'' \sqsubseteq_{F \cup F'}^{W \cup W'} H$ 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 $G'' \sqsubseteq_{F \cup F'}^{W \cup W'}$. 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 suppose 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 restricted to the vertex set, acyclic, and properly typed, and that is has one edge to every vertex input. It is restricted to the vertex set since all edges involving W and W' were removed and none were added. It is acyclic. It is properly typed since every edge is properly typed in G or G', and the types of the vertices have not changed. Finally, it 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. □

The two graphs of Figure 3 actually stand in a much more demanding relation: executions of the two graphs on the same value always agree.

Definition 7 *Suppose $G = (V, E)$ and $G' = (V', E')$ are trace graphs. $G' \boxed{\sqsubseteq} G$ if and only if:*

- $G' \sqsubseteq G$. *(This defines an injective function h from vertrex inputs and outputs of V' into corresponding vertex inputs and outputs of V.)*
- $\mathrm{dom}(G) = \mathrm{dom}(G')$.
- *For any x on which G has execution f and G' has execution f', $f' = f \circ h$.*

$G' \sqsubset_c G$ (read "G is conservatively thinner than G") if and only if $G' \sqsubseteq_c G$ and G' is not isomorphic to G.

A conservative thinning removes vertices from a graph in a way that preserves the values assigned to the remaining vertices by all executions. 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 8, 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 3 lists a variety of other properties of $\sqsubseteq_c$.

Theorem 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).*
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 property follows directly from the definition. Reflexivity follows because $\sqsubseteq$ is reflexive and, by Theorem 1, there is at most one execution of a trace graph on any input. The identitive property is a consequence of the identitive property of $\sqsubseteq$. The transitive property also 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 $G'' \sqsubseteq_c G$. □

3.1.4 The PLC for Trace Graphs

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

Definition 8 (PLC—Trace Graphs) *For a trace graph G there should be no $G' \sqsubset_c G$.*

Compare this definition with the informal statement of the PLC in Definition 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.

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 4 *Consider the class of trace graphs containing only two-input integer additions, multiplications and exponentiations, positive integer constants, and integer equality predicates. 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 iff the program with Gödel number N does not halt on any input [JM84]. □

Of course, when the functions in the trace graph have finite domains the question is decidable, though it may well be intractable.

3.1.5 Discussion

Trace graphs, executions, and thinnings serve here as tools for exploration of the PLC. It might be possible to give a formal semantics for a functional language in terms of trace graphs; recursive program semantics have been studied for structures similar to trace graphs, for example by Arbib [AM79].

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.

A trace graph has a functional type induced by the typing of the vertices. The details of the type system are unimportant; what matters is that runtime type errors must be disallowed in order to achieve complete locality of effect.

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], rather than the more traditional partial evaluation with respect to fixed inputs as in the work of Ershov [Ers82]. 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.

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

3.2 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 take conditional execution into account, but still defer issues relating to recursive structures in the source language.

3.2.1 Sets of Trace Graphs

. To represent a program, a trace graph set must meet several conditions. Of course, the types 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 dom(S) to denote $\bigcup_{G \in S} \mathrm{dom}(G)$. The set must also be deterministic, in the sense that it has no 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 4. It has exactly one graph with an execution for any input, but the difference between the two graphs in the set represents a choice that must be made by any corresponding program, and there is no predicate that can be evaluated in both traces in time to guide the choice. Any program with these two traces would have to be prescient—it would have to decide immediately and spontaneously whether to compare or to subtract.

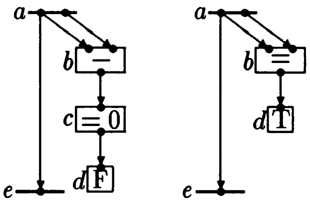


Figure 4: A non-executable set of trace graphs

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 9 *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 P_T or P_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 10 *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 leaf giving S. Otherwise, a decision tree for S is a node giving both S and a predicate partition for S, whose right child is a decision tree for that subset of S for which the predicate is true, and whose left 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 11 *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 considered as a kind of program in itself, since it contains exactly one trace graph with an execution for any input.

Theorem 5 *If S is an executable set of trace graphs 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. 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 $f(v_{in}(1)) =$ true and $f'(v'_{in}(1)) =$ false. But v and v are matching vertices, so $f'(v'_{in}(1)) = f(v_{in}(1))$, which is a contradiction. □

Theorem 5 shows that any executable set is deterministic. As Figure 4 shows, the reverse is not true.

3.2.2 The Thinning Relations for Trace Graph Sets

The idea of thinning developed for individual trace graphs extends naturally to sets of trace graphs. One set of trace graphs is thinner than another when every member of the thinner set is thinner than some member of the thicker set. That is:

Definition 12 *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 sets in S' to isomorphic sets in S.*

Theorem 6 summarizes some properties of the $\sqsubseteq$ relation on sets.

Theorem 6 *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 $G'' \sqsubseteq S$. 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 $S \subseteq S$. □

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 $\sqsubseteq_c$ relation also generalizes to trace graph sets, but more care is required. Obviously, a conservatively thinner set should be, structurally, a thinner set; just as with individual trace graphs, the additional constrait has to do with agreement between executions. A trace graph set is a function just like an individual trace graph (except that it may be non-deterministic). 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 13 *Suppose* S *and* S' *are sets of trace graphs.* $S' \sqsubseteq_c S$ *if and only if:*

1. $S' \sqsubseteq S$.

2. $\mathrm{dom}(S') = \mathrm{dom}(S)$.

3. *For any execution* f' *of any* $G' = (V', E') \in S'$ *on any input* x, *there is an execution* f *of some* $G = (V, E) \in S$ *on* x, *such that* G' *is isomorphic to a thinning of* 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 7 summarizes some other properties of the $\sqsubseteq_c$ relation for sets.

Theorem 7 *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'' on some x, there is a thicker $G' \in S'$ with an execution f' that agrees with f'', and so 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 one 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 that 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 5. 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.

3.2.3 The PLC for Executable Sets

The principle of least computation for individual trace graphs expressed in Definition 8 extends naturally to executable sets of trace graphs.

Definition 14 (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 14. 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 6. There are several non-isomorphic sets conservatively thinner than the set $\{G, H\}$ shown—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.

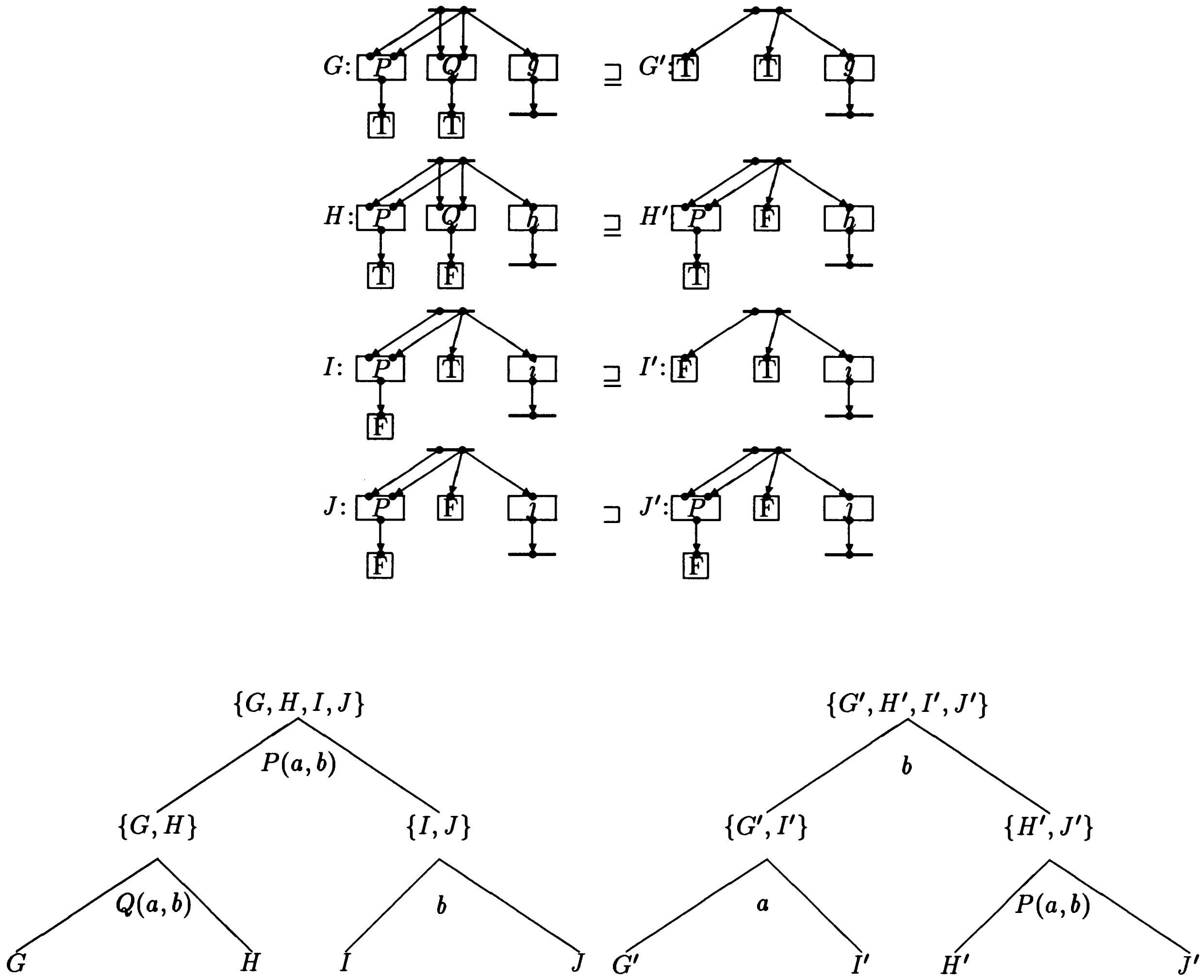


Figure 5: Trace graph sets $\{G, H, I, J\} \sqsupseteq \{G', H', I', J'\}$, with decision trees

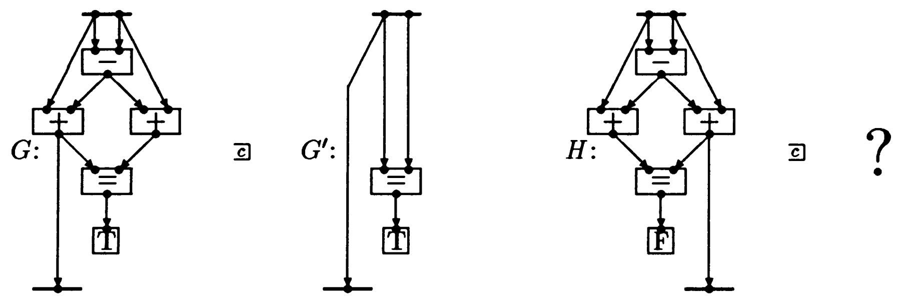


Figure 6: A set $\{G, H\}$ not quite inviolation of the PLC

3.2.4 Discussion

There is an interesting connection between sets of trace graphs that are not necessarily executable and non-deterministic 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 it a good domain for the formal expression of the PLC and the idea of thinning. In the next section we will see that recursive structure makes these simple concepts much harder to formalize.

Executable sets also 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. The examples in Section 4 should clarify this point.

3.3 Trace Grammars and the PLC

Executable sets of trace graphs are potentially infinite, since a program with infinitely many different possible paths generates an infinite set of trace graphs. Out next goal is to purge these infinite sets from the discussion and arrive at a finite representation, comparable to ordinary functional programs. We therefore turn to trace grammars, which are finite objects that generate sets of trace graphs.

3.3.1 Trace Grammars

Trace grammars will provide a mechanism for deriving one trace graph from another, and so for generating a language of trace graphs from an original "start graph". Trace grammars are a specialization of the directed, node-label controlled (DNLC) graph grammars studied by Janssens and Rozenberg [Roz87]. In a such a 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 inputs and outputs. Remove the mother vertex from the host graph and remove the input and output vertices from the daughter graph, then stitch the daughter graph into the host graph with a "seam" of edges connecting the matching inputs and outputs. This seam of edges includes one edge to each vertex input left sourceless in the daughter graph (those that used to have an edge from the input vertex of the replacement 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 15 *Let $G = (V, E)$ and $H = (W, F)$ be trace graphs with disjoint vertex sets, and let v be a vertex in V_C whose input vector has a type matching the output vector of W_i and whose output vector has a type matching the input vector of W_o. Define E_{old} to be E restricted to $V - \{v\}$, 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 $w_{in}(k)$, E_{seam} has an edge $(u_{out}(i), w_{in}(k))$, where $u_{out}(i)$ was the source of the edge in E to the jth vertex input of v.*
- *For each edge from v, $(v_{out}(j), u_{in}(k)) \in E$, E_{seam} has an edge $(x_{out}(i), u_{in}(k))$, where $x_{out}(i)$ 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 $V' = V - \{v\} \cup W_C \cup W_T \cup W_F$ 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 patched in in place of a vertex: just connect the matching inputs and outputs. Figure 7 shows an example of a vertex replacement.

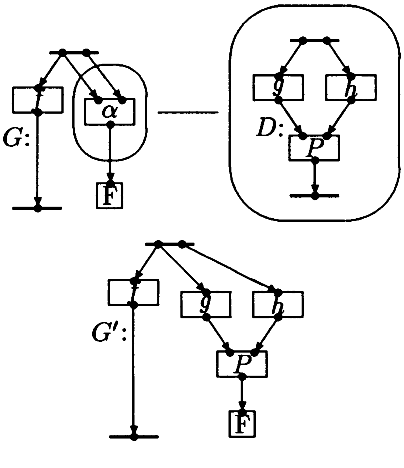


Figure 7: The replacement of a vertex in G by a graph D

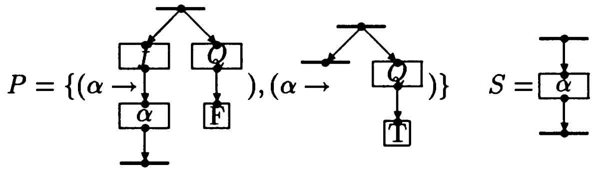
Definition 16 *A* trace grammar *is a 4-tuple (N, T, P, S). N is a finite set of* non-terminal functions *and T is a finite set of* terminal functions. *(N and T are disjoint.) P is a finite set of* productions *of the form $(\alpha \rightarrow D)$, where $\alpha \in N$ and D is a trace graph with inputs and output matching those of α. S is a graph called the* start graph. *All of the vertices in P and S have functions in $N \cup T$.*

Suppose that $Q = (N, T, P, S)$ is a trace grammar and $G = (V, E)$ and $G' = (V', E')$ are trace graphs in which all of the vertices have functions in $N \cup T$. Then $G \Rightarrow_Q G'$ if and only if there is a vertex $v \in V_C$ with a non-terminal function 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 deriviations from G to G'. The $\Rightarrow_Q^*$ relation can be used to define the set of trace graphs generated by a trace grammar:

Definition 17 *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.*

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

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



The first three elements of the generated set:

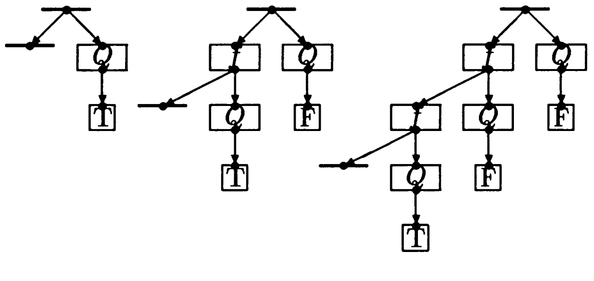


Figure 8: Example of a trace grammar and generated set

Figure 8 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 like that of Figure 8 that obey this restriction are *normal* trace grammars.

Definition 18 *A* normal *trace grammar $Q = (N, T, P, S)$ is one in which all non-terminals α in N have the property that $\{D \mid (\alpha \rightarrow D) \in P\}$ is an executable set.*

Theorem 8 *If Q is a normal trace grammar, $L(Q)$ is executable.*

Proof: Let R be the set generated by a normal trace grammar $Q = (N, T, P, S)$. Define a function f that maps each graph $G \in R$ to a breadth-first derivation of G.[2] 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 non-terminal, and since f chooses a breadth-first derivation, that non-terminal has terminal ancestors only. So $T_f(R)$ can start with a decision tree for the possible expansions of that non-terminal, 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, attach the decision tree for $T_f(R')$. Since all $f(G)$ for $G \in R'$ agree on the first $n+1$ steps, $T_f(R')$ will only use predicates generated by non-terminals that were not yet expanded when $T_F(R)$ was constructed. It follows that no predicate is used more than once, so the resulting structure is a decision tree for R. □

3.3.2 The Correspondence Between Trace Grammars and CFGs

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 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 M be a type. Let P_i denote a distinct function of type $(M \rightarrow \text{bool})$ 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 the same pattern of truth assignments. Finally, let f^x denote a distinct function of type $M \rightarrow M$ for each symbol x in Σ^+. The graph component corresponding to a symbol x in Σ^+ is as pictured in Figure 9. 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 are all of type M and they will have a common source as indicated in the diagram. The output of the component is the output from f^x, also of type M.

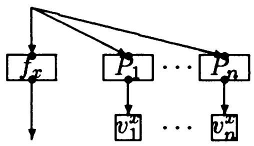


Figure 9: The trace graph component corresponding to $x \in \Sigma^+$

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

[2]A trace graph does not necessarily have a unique breadth-first derivation. The axiom of choice is not required for proof, however, since derivations can be ordered and the least breadth-first derivation selected for each graph.

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 of type M 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, which has type M.

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 9 *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^ϵ 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. □

Theorem 10 *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 with non-terminal 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 non-terminals: a non-terminal in a string corresponds to a non-terminal function in the graph. N' contains a non-terminal function f^V of type $(M \rightarrow M)$ for every $V \in N$. T' contains a terminal function f^x of type $(M \rightarrow M)$ for every $x \in T$. Now for every production $(V \rightarrow s_1 \ldots s_n) \in P$, P' contains a production $(f^V \rightarrow G)$ where G is the graph corresponding to the string $s_1 \ldots s_n$. Finally, start graph S' is simply the graph corresponding to the string S (so it has a non-terminal function f^S followed by the component for ω). By construction, if G is a graph corresponding to a string r, $G \Rightarrow_Q G'$ if and only if G' corresponds to a string r' that derives from r by one application of a production in the CFG. Since the start graph S' corresponds to the string

S, it follows that $S \Rightarrow_Q^* G$ if and only if the string corresponding to G derives from S in the CFG. Hence, a set generated by Q corresponds to the language generated by the CFG.

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 also assume that the given trace grammar has no useless non-terminals (that is, that all non-terminals can be expanded into terminal graphs). This is without loss of generality since a trace grammar with useless non-terminals can easily be converted into one without them, by essentially the same method that removes useless non-terminals 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 non-terminals. Define a CFG $C = (N, T, P, S)$ as follows. N contains a non-terminal symbol α_j^i for each non-terminal 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 in $T \cup N$ as follows: each terminal function f^x corresponds to an x in the string, each non-terminal function α entered in the path at input i and exited at output j corresponds to an α_j^i in the string, and anything else in the path is not reflected in the string. For each production $(\alpha \rightarrow D) \in P'$, 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_j^i 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' is not of type $(M \rightarrow M)$ 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.) We will show that the language of C corresponds to the set generated by Q.

If a string (possibly including non-terminal symbols) other than S is derivable in C then it corresponds to a path through some graph derivable in Q. Proof is by induction on the length n of the derivation in C. The base case is $n = 1$: every string derivable by one step from the start symbol S corresponds a path through the start graph S'. For the inductive case, suppose some string r is derivable in C by $n > 1$ steps. The previous string r' in the derivation of r corresponds, by the inductive hypothesis, to a path through some graph G' derivable in Q. The final step from r' to r is the application of a production mapping some α_j^i to a substring. By construction of P that substring corresponds to a path from the i'th input to the j'th output in some graph D with a production $(\alpha \rightarrow D) \in P'$, so by applying that production to the corresponding α in G' we get a graph G with a path corresponding to r.

Similarly, if a graph is derivable in Q than any path through that graph (possibly including non-terminal functions) corresponds to some string derivable in C. Proof is by induction on the length n of the derivation in Q. The base case is $n = 0$: every path through the start graph S' corresponds to the string r in a production $(S \rightarrow r) \in P$. For the inductive case, suppose some graph G is derivable in Q by $n > 0$ steps, and let q be any path through G. The previous graph G' in the derivation of G contains a precursor path q': the same path as q but possibly having a non-terminal function in place of some subpath. By the inductive hypothesis path q' corresponds to some string r' derivable in C. If $q' \neq q$ then the final step from G' to G expands some non-terminal α in path q'. This α is connected in q' by some input i and output j, and the final step expands α into a graph D with a path from i to j. It follows that there is a production in P that expands α_j^i into a string corresponding to that path in D, and by applying this production to the corresponding α_j^i in r' we get a string r corresponding to path q.

The final step in the proof is to show that the language of the CFG C corresponds to the set generated by the trace grammar Q. Suppose a string s is in the language of C. Then, as shown above, Q derives a graph G with a path of terminal functions from input to output corresponding to s. We cannot assume that G itself is terminal, but since Q contains no useless symbols, a terminal graph G' can be derived from G. This G' is in the set generated by Q, so 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. Or suppose G is one of the (terminal) graphs generated by Q. Since G corresponds to a string, it has exactly one path from input to output; and as shown above, C 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 M corresponds to some set of trace grammars S in which no two elements are isomorphic to each other. By Theorem 9, S is executable. But by Theorem 10, S has a trace grammar only if M is context-free. So not every executable set has a trace grammar: if M is not a context-free language, the corresponding set S 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?

3.3.3 Thinning and Trace Grammars

The $\sqsubseteq$ and $\sqsubseteq_c$ relations extend to graph grammars in the obvious way: one grammar is thinner than another when the langauge it generates is thinner.

Definition 19 *The relations $\sqsubseteq$, $\sqsubset$, $\sqsubseteq_c$ and $\sqsubset_c$ hold for graph 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 10 shows that the $\sqsubseteq_c$ relation is undecidable even for individual trace graphs, and with trace grammars there are, in addition, all the usual undecidabilities of fully expressive formalisms for computation. For example, a grammar generating the empty set is conservatively thinner than a grammar generating a set S if and only if $\mathrm{dom}(()S) = \emptyset$, so an effective characterization of $\sqsubseteq_c$ would decide the halting problem.

What about $\sqsubseteq$? It is a relatively simple relation on graph structures and makes no reference to executions. One potentially useful characterization would be as a class 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 6 and 10. 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 even decidable? We 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 $\mathrm{Omit}(L)$ 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 $L(C) \subseteq \mathrm{Omit}(L(C'))$? Using the construction of Theorem 10 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: $\mathrm{Omit}(L(C'))$ is a regular language.[3] Perhaps the $\sqsubseteq$ relation is decidable too.

3.3.4 The PLC for Trace Grammars

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

Definition 20 (PLC—Executable Trace Grammars) *For an executable trace grammar Q there should be no executable $Q' \sqsubset_c Q$.*

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.

As discussed above, this is prodigously undecidable. Since one goal of this theory is to support a practical optimizer based on the PLC, we need to identify some classes of PLC-violation that are decidable. About the most we can hope to accomplish is to identify violations of the PLC for finite trace graph sets with finite domains. Given such a set it is possible either to generate a conservatively thinner set, or to prove that none exists.

Here's a way to make use of this ability: gather the set of right-hand sides of productions for a single non-terminal, optimize that, and then modify the grammar accordingly. The one thing that stands in the way is that the right-hand sides may contain non-terminal vertices, so the $\sqsubset_c$ relation is not defined for them. It could be defined with reference to expansions of the non-terminals, but this leads back to infinite sets and undecidable problems. To localize the problem one must treat non-terminals more like terminals, without investigating how they expand.

The solution is to test $\sqsubset_c$ relative to a mapping from non-terminals to partial functions. If H and G are trace graphs containing non-terminals and $\mathcal{F}$ is a function mapping each non-terminal to a function of the appropriate type, and if that mapping gives $H \sqsubset_c G$, we'll say that *$H \sqsubset_c G$ relative to $\mathcal{F}$*.

[3] A neat proof that $\mathrm{Omit}(L(C'))$ 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.

Now there are two approaches to thinning right-hand sides of productions. Strictly speaking, one should assume nothing about $\mathcal{F}$: any non-terminal 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 non-conservative. Since the domain of the arbitrary function associated with a non-terminal is unknown, a conservative thinning must retain at least one computation of each non-terminal function on each distinct input value. Treating non-terminal 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.

Theorem 11 *Let $Q = (N, T, P, S)$ be a normal trace grammar. For some non-terminal α 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 any mapping $\mathcal{F}$ from non-terminals 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 $D' \in A'$. Then $Q' \sqsubseteq Q$.*

Proof: Because Q is normal each non-terminal corresponds to a partial function, obtained as the executable set generated from that non-terminal under Q. Define $\mathcal{F}$ to be this mapping from non-terminals to their functions under Q.

Suppose we have a graph G and set R, possibly containing non-terminals, for which $\{G\} \sqsubseteq R$ relative to $\mathcal{F}$, and suppose G derives some terminal graph I in Q'. Then R derives some terminal set T in Q for which $\{I\} \sqsubseteq T$. Proof is 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. Every graph in R generates a subset of S which computes $\mathcal{F}(\beta)$ for every non-terminal β in the graph. So, since $\{I\} \sqsubseteq R$ relative to $\mathcal{F}$, $\{I\} \sqsubseteq S$. 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 non-terminal $\beta \neq \alpha$, 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 $\mathcal{F}$, and since the corresponding β has been replaced with a function equal to $\mathcal{F}(\beta)$ (though perhaps on a smaller domain than that of $\mathcal{F}(\beta)$), $\{H\} \sqsubseteq S$ relative to $\mathcal{F}$. 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 $\mathcal{F}$ and since $\{D'\} \sqsubseteq A_{D'}$ relative to any mapping, $\{H\} \sqsubseteq S$ relative to $\mathcal{F}$. Now since $H \rightarrow^*_{Q'} I$ by $n-1$ steps, by the inductive hypothesis S derives some terminal set T in Q for which $\{I\} \sqsubseteq T$.

For the start graph S we have $\{S\} \sqsubseteq \{S\}$ relative to $\mathcal{F}$, so by the result above we can conclude that for any G generated by Q' there is a set T generated by Q for which $\{G\} \sqsubseteq T$. We can also conclude that anything in the domain of $L(Q')$ is in the domain of $L(Q)$. It remains to be shown that the the reverse is also true.

If x is an input vector, G is graph possibly containing non-terminals, I is a terminal graph with an execution on x, and $G \rightarrow^*_Q I$, then $G \rightarrow^*_{Q'} J$ for some terminal graph J with an execution on x. Proof is by induction on the length n of the derivation of I. If $n = 0$, $G = I = J$ and we're done. If $n > 0$, let H be the result of the first step in the derivation if I, so $G \rightarrow_Q H \rightarrow^*_Q I$. If that first step is the expansion of some non-terminal $\beta \neq \alpha$, $G \rightarrow_{Q'} H$ and then, by the inductive hypothesis, $H \rightarrow_{Q'} J$ for some terminal graph J with an execution on x. So suppose the first step is the application of some production

$(\alpha \to D)$. Since $G \to_Q^* I$, we know that G has an execution on x relative to $\mathcal{F}$. There is some production $(\alpha \to D')$ in Q' where D' has executions on everything D does relative to any mapping. So application of that production to G yields a graph H with an execution on x relative to $\mathcal{F}$. But then $H \to_Q^* I'$ for some terminal graph I' with an execution on x, so, by the inductive hypothesis, $H \to_{Q'}^* J$ for some terminal graph J with an execution on x.

So for any x, if Q generates a graph with an execution on x, Q' does too. Now we know that $L(Q)$ and $L(Q')$ have the same domains, and that for any $G' \in L(Q)$ there is some $R \subseteq L(Q)$ for which $\{G'\} \sqsubseteq R$. It follows that $Q' \sqsubseteq Q$. □

Another, more reckless approach is to treat each non-terminal as an arbitrary, deterministic, *total* function. This is not a safe way to approximate full conservative thinning, but it is very common in practice. It has the effect of potentially enlarging the set of inputs on which the program terminates, but it is conservative otherwise.

3.3.5 Discussion

Consider how a trace grammar expresses non-termination. 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 non-terminal trace graph; once again, no infinite graph is generated, and there is no trace graph for those inputs on which the program diverges. A trace grammar for a program that diverges on an input generates an executable set containing no graph with an execution for that input—an executable set with a decision tree that is not a full binary tree.

The fact that the language of a program's paths of execution can be generated by a context-free (graph) grammar is at first a bit alarming. It is known that the set of legal executions of a Turing machine can be expressed as the intersection of two context-free languages [HU79]; but to assert that it can be generated by a single context-free grammar would be disastrous since, for example, the question of whether the language is empty is decidable for CFG's and undecidable for TM's. Closer examination shows that there is no cause for alarm: an executable set of trace graphs only captures structural properties of a program's paths of execution, not the program's actual behavior on inputs. If a program loops on every path, the corresponding executable set is empty and this can be determined by inspection of the trace grammar. But when a program has the structure to halt, the question of whether it ever actually does so is still up in the air, as it must always be.

4 Examples of Conservative Thinning

The previous section developed the trace graph formalism and a rigorous statement of a well-known intuitive principle of optimization, the PLC. One of the strengths of this formalism is that it "brings out the worst" in programs: if a program violates the PLC, this fact is manifest in the trace graph set. Many well-known optimizations are conservative thinnings whose fundamental similarity is hidden by other representations but revealed by the trace graph representation.

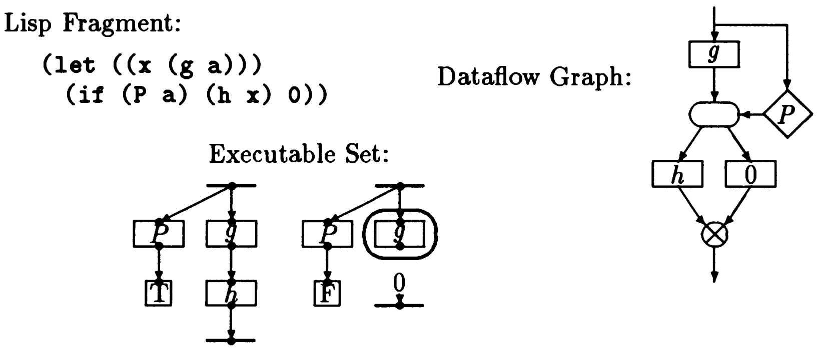


Figure 10: Comparison of a dataflow graph and an executable set

Figure 10 compares three representations of a program: a Lisp expression, an executable set, and a dynamic dataflow graph in the style of Arvind [AN87]. The program 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 executable set representation makes it clear that this optimization merely removes an intermediate computation (the circled vertex) without affecting any other values: in short, it is a thinning, much like dead code removal. Figure 11 shows another such comparison; again, the dataflow graph representation obscures an obvious thinning.[4]

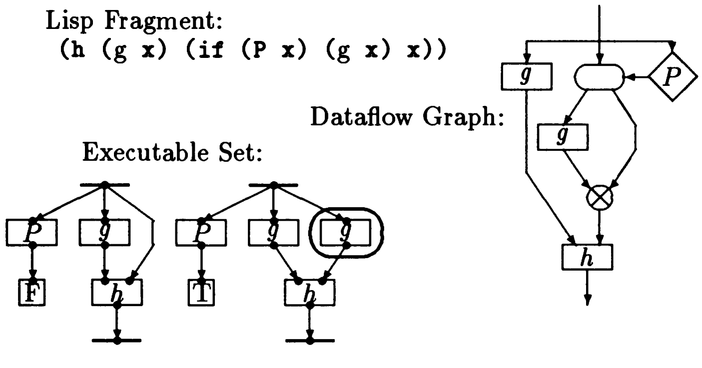
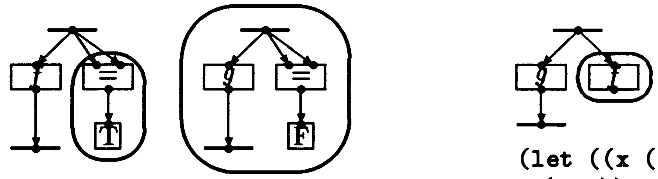


Figure 11: Another dataflow graph comparison

Dead code removal is also a special case of conservative thinning. This may seem counterintuitive at first since the PLC says nothing about the size of the code. However, if code is dead in the sense that one arm of a conditional is never exercised, there is a predicate

[4] 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.



`(if (= a a) (f a) (g a))`

```
(let ((x (f a)))
  (g a))
```

Figure 12: Dead code removal

in the executable set which is unnecessary according to PLC and can be removed; and if code is dead in the sense that it computes a value of which there is no subsequent use, that computation is unnecessary. Both of these situations are shown in Figure 12.

Figure 13 shows an example of the removal of loop-invariant computations. The diagram shows the first four members of the executable set for a program like the Lisp program indicated. The unnecessary repeated computations of the function f circled in the diagram can be thinned out.

Loop jamming is also a conservative thinning. Figure 14 shows two loops which can be jammed. Note that in this case, the trace grammar also generates many graphs without executions—all those in which one loop terminates before the other. The trace graphs shown are the first three from the generated set that do have executions.

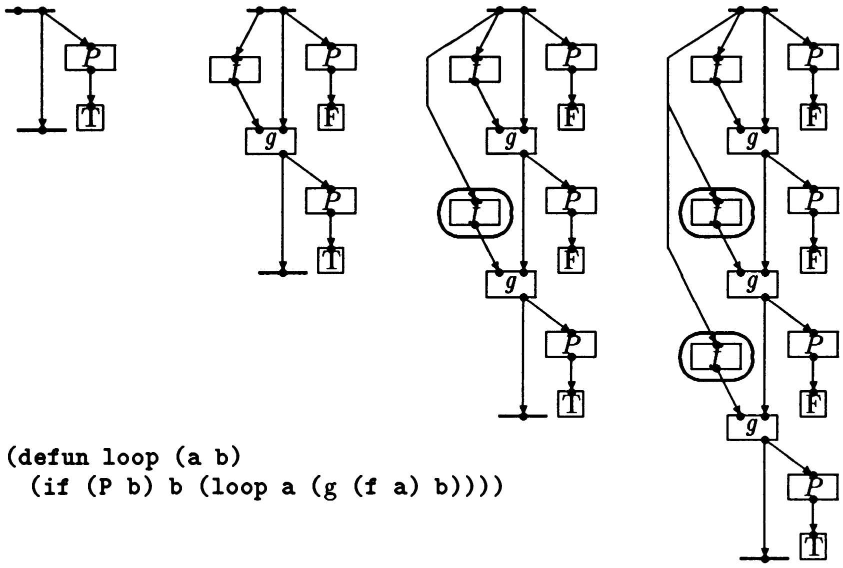
Not all common compiler optimizations are PLC-inspired. Strength reduction, for example, exchanges one primitive function for another, while thinning cannot introduce a new function and has no model of the comparative cost of primitives. Some algebraic transformations are conservative thinnings, but others are not: for example, `(+ (* a b) (* a c))` can be transformed by the distributive law to `(* a (+ b c))`, but this is cannot be a thinning since the value `(+ b c)` never occurred in the original computation.

5 Preliminary Experiments

When a traditional compiler optimizes a program it begins by deriving relations among program parts. It establishes, for example, that two parts compute equivalent expressions, or that there is a path from one part to another. When the compiler has exhaustively generated the relations of interest it uses them to evaluate the preconditions of various canned program transformations.

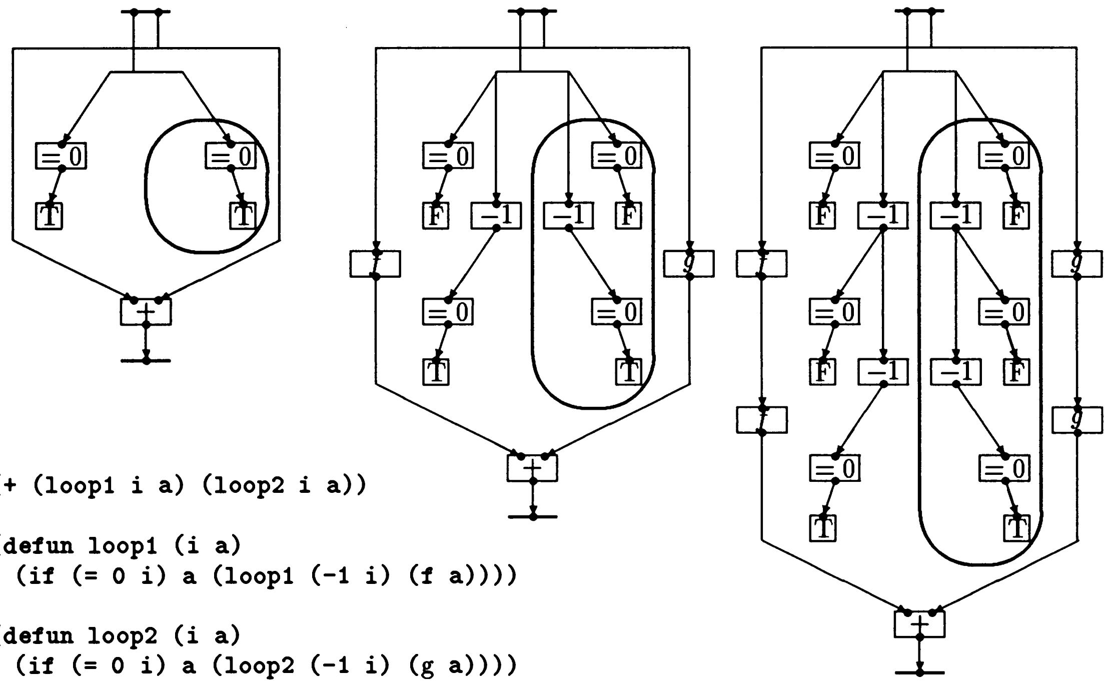
In a more general optimizer the relations derived in support of optimization ought to be relations among the values that will occur when the program is executed, not relations among static program parts, and the transformational competence ought to be fine-grained and flexible, not limited to a small repertoire of coarse patterns. Enter, trace graphs and thinning: a representation that supports reasoning about individual run-time values, and a fine-grained and flexible transformation that corresponds to broad principle of optimization.

We have begun experimenting with an optimizer based on thinning. This program, THINNER, currently operates on a small strongly-typed subset of Lisp. The first task of THINNER is to compile the input program into a trace grammar. This is actually quite simple: we associate a non-terminal function with each function in the input program, and compile a normal trace grammar (one with an executable set of trace graphs for each non-terminal).




```
(defun loop (a b)
  (if (P b) b (loop a (g (f a) b))))
```

Figure 13: Loop invariant removal




```
(+ (loop1 i a) (loop2 i a))

(defun loop1 (i a)
  (if (= 0 i) a (loop1 (-1 i) (f a))))

(defun loop2 (i a)
  (if (= 0 i) a (loop2 (-1 i) (g a))))
```

Figure 14: Loop jamming

The second task of THINNER is to discover properties of the arcs in graphs in the grammar. Each arc in the right-hand side of a production represents a value that may occur when the function corresponding to the non-terminal on the left-hand side is evaluated. THINNER uses a simple axiomatization of each terminal function in the graph to discover arc properties, including equality and inequality, integer-$\leq$, and expressions for arc values in a canonical form. THINNER's special-purpose inference engine proves as much as it can about the execution of the program in the absence of actual inputs. Its ultimate aim is to discover cases of arc equality: when the value assigned to one arc is equal to the value assigned to another for every execution of the given graph. It also discovers contradictions, which indicate that the graph in question has no executions and can be removed during thinning.

Using the arc equalities it has discovered, THINNER looks for executable thinnings of the graph grammar. Obviously, the only provably-correct bypass arcs are those which substitute equals for equals. In addition, the THINNER must verify that the proposed conservative thinning preserves the normality of the grammar—that is, that it preserves the executability of the set of right-hand sides for the non-terminal in question. This may require thinning several graphs in the set simultaneously.

The most difficult task for the THINNER (and the one with which we are experimenting the most heavily) is the task of folding and unfolding non-terminals. We unfold to expose more graph structure to thinning, and we fold to make use of thinned graphs recursively. This is the only operation that really treats the grammar structure of the graph; the inference engine reasons about individual graphs and the thinner operates on executable sets of graphs (as in the examples of Section 4).

Figure 15 shows some transformations performed by the current version of THINNER. We stress that these optimizations—dead code elimination, loop invariant removal, and the optimization of an exponential-time Fibonacci function into a linear-time version—were not applied by rote, as an ordinary optimizer might do, but discovered by direct application of the Principle of Least Computation.

5.1 Conclusions

We have presented a new approach for the automatic optimization of computer programs: deriving optimizations as needed from broad intuitive principles. One principle, the Principle of Least Computation, and one domain, the domain of purely functional first-order programs, were explored, and we derived a formal statement of the principle for that domain in terms of conservative thinning of trace grammars. This is the theoretical basis of our work. We also gave examples and preliminary experimental results indicating that our statement of the PLC is workable and that our goal of automatically deriving optimizations from it is attainable.

Work is in progress to develop a full thinning-based optimizer and to evaluate its performance on a variety of optimizing problems. We do not seek to out-perform ordinary optimizers on traditional compiler optimizations; indeed, it would be foolish for a program to "rediscover" loop invariant removal over and over. The canon of traditional compiler transformations will always be the first resort of the practical optimizer. With further advances in principled optimization, it need not always be the last.

[5] Elapsed time measured on a Sun 4, rounded up to the nearest second.

input	*output*	*time*[5]
`(defun id (a)` `(if t a (- a 1)))`	`(defun id (a)` `a)`	1s

`(defun loop (a b)`
`(if (< 2 a)`
`(loop (+ a (* b (* b b)))`
`b)`
`a))`
	`(defun loop (a b)`
	`(if (< 2 a)`
	`(let ((c (* b (* b b))))`
	`(loop2 (+ a c) b c))`
	`a))`
	`(defun loop2 (a b c)`
	`(if (< 2 a)`
	`(loop2 (+ a c) b c)`
	`a))`
		9s

`(defun fib (a)`
`(if (< a 2)`
`a`
`(+ (fib (- a 1))`
`(fib (- a 2)))))`
	`(defun fib (a)`
	`(if (< a 2)`
	`a`
	`(multiple-value-bind (x y)`
	`(fib2 (- a 1))`
	`(+ x y))))`
	`(defun fib2 (b)`
	`(if (< b 2)`
	`(values b 0)`
	`(multiple-value-bind (x y)`
	`(fib2 (- b 1))`
	`(values (+ x y) x))))`
		16s

Figure 15: Some preliminary experiments with THINNER

6 Acknowledgement

The author gratefully acknowledges the advice and encouragement of Devika Subramanian, who proposed the problem treated in this paper.

References

[AM79] M. A. Arbib and E. G. Manes. Partially-additive monoids, graph-growing and the algebraic semantics of recursive calls. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology.* Springer-Verlag, 1979. LNCS 73.

[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.* Springer-Verlag, March 1985. LNCS 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.

[Ber90] Andrew A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming.* ACM, June 1990.

[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*, 1977.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3), July 1983.

[Ers82] A. P. Ershov. Mixed computation: potential applications and problems for study. *Theoretical Computer Science*, 18, 1982.

[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.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JM84] J.P. Jones and Y.V. Matijasevic. Diophantine representation of the set of prime numbers. *Journal of Symbolic Logic*, 49, 1984.

[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.

[Per89] Mark Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[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.

[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. LNCS 291.

[RW88] Charles Rich and Richard C. Waters. A research overview. *Computer*, 21(11), November 1988.

[SG87] Devika Subramanian and Michael R. Genesereth. The relevance of irrelevance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[Sub89] Devika Subramanian. *A Theory of Justified Reformulations*. PhD thesis, 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*, 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*. Springer-Verlag, 1980. LNCS 85.

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), July 1986.