Solving Regular Tree Grammar Based Constraints*

Yanhong A. Liu Ning Li Scott D. Stoller

July 2000 (Revised October 2000)

Abstract

This paper describes the precise specification, design, analysis, implementation, and measurements of an efficient algorithm for solving regular tree grammar based constraints. The particular constraints are for dead-code elimination on recursive data, but the method used for the algorithm design and complexity analysis is general and applies to other program analysis problems as well. The method is centered around Paige's finite differencing, i.e., computing expensive set expressions incrementally, and allows the algorithm to be derived and analyzed formally and implemented easily. We study higher-level transformations that make the derived algorithm concise and allow its complexity to be analyzed accurately. Although a rough analysis shows that the worst-case time complexity is cubic in program size, an accurate analysis shows that it is linear in the number of live program points and in other parameters, including mainly the arity of data constructors and the number of selector applications into whose arguments the value constructed at a program point might flow. These parameters explain the performance of the analysis in practice. Our implementation also runs two to ten times as fast as a previous implementation of an informally designed algorithm.

1 Introduction

Regular tree grammar based methods are important for program analysis, especially for analyzing programs that use recursive data structures [25, 36, 20, 50, 29]. Basically, a set of grammar-based constraints is constructed from the program and a user query and is then simplified according to a set of simplification rules to produce the solution. Usually, the constraints are constructed in linear time in the size of the program, and the efficiency of the analysis is determined by the constraint-simplification algorithms.

This paper describes the precise specification, design, analysis, implementation, and measurements of an efficient algorithm for solving regular tree grammar based constraints. The particular constraints are for dead-code elimination on recursive data, but the method used for the algorithm design and complexity analysis is general and applies to other program analyses as well.

The method is centered around Paige's finite differencing [38, 41, 39], i.e., computing expensive set expressions incrementally. It starts with a fixed-point specification of the problem, then applies (1) dominated convergence at the higher level [10] to transform fixed-point expressions into loops, (2) finite differencing [41, 39] to transform expensive set expressions in loops into incremental operations, and (3) real-time simulation at the lower level [40, 9] to transform sets and set operations to use efficient data structures. This method allows the algorithm to be derived and analyzed formally and implemented easily.

We first give a precise fixed-point specification of the problem. We then transform it into a loop and apply finite differencing completely systematically, making all the steps explicit. At the higher level, we study new transformations that make the derived algorithm concise and allow its complexity to be analyzed accurately. The complexity analysis captures the exact contribution of each parameter. In particular, although a rough analysis shows that the worst-case time complexity is cubic in program size, an accurate analysis shows that it is linear in the number of live program points and in other parameters, including mainly the arity of data constructors and the number of selector applications into whose arguments the value constructed

*This work is supported in part by ONR under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109, by NSF under grants CCR-9711253 and CCR-9876058, and by a Motorola University Partnership in Research Grant. Authors' address: Y.A. Liu and S.D. Stoller, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794; Email: {liu,stoller}@cs.sunysb.edu; Phone: (631)632-{8463,1627}; Fax: (631)632-8334; N. Li, Computer Science Department, University of Wisconsin at Madison, Madison, WI 53706; Email: ning@cs.wisc.edu

at a program point might flow. These parameters explain the performance of the analysis in practice. At the lower level, we show that real-time simulation using based representation [40] applies only partially to our application, and we discuss data structure choices and the trade-offs. In particular, our accurate complexity analysis at the higher-level suggests that combination with unbased representation works well in our application, and our experiments support this. Our implementation runs two to ten times as fast as a previous implementation of an informally designed algorithm [29].

The main contributions of this work are

(1) the application of a powerful, systematic transformational design methodology that leads from a precise high-level fixed-point specification of a nontrivial problem to a highly efficient algorithmic solution,

(2) the identification of parameters in problem instances and the precise expression of the algorithm complexity in terms of these parameters, and

(3) the implementation and experiments that help confirm the accuracy of the complexity analysis and compare the efficiency of the algorithm with that of an informally designed algorithm.

It is not the goal of this paper to show a drastically new algorithm or algorithm design method. Instead, since program analysis is a central recurring task in compiler construction, the goal is to show the systematic nature of the design method in the hope that it can be more widely used for developing analysis algorithms, to allow easier correctness proof, algorithm understanding, performance analysis and comparison, as well as implementation. At the same time, through such usage, one may further improve the design method, for example, as we study the transformations and accurate complexity analyses enabled by Theorem 5.1.

The rest of the paper is organized as follows. Section 2 specifies the problem. Section 3 gives an overview of the algorithm design and analysis approach. Sections 4, 5, and 6 describe finite differencing applied to the derivation of our simplification algorithm, higher-level design and analysis, and lower-level implementation and experiments, each with accurate complexity analysis. Section 7 discusses complexity and other issues. Section 8 discusses related work and concludes.

2 Problem specification

The specification from the application. We first look at the grammar constraints and the simplification algorithm for the dead-code elimination application in [29][1]. There, regular tree grammars, called liveness patterns, represent projection functions that project out components of values and parts of programs that are of interest.

The grammar constraints constructed from a given program or given in a user query consist of productions of the following standard forms:

$N \rightarrow d$	dead form,	where d is a special constant
$N \rightarrow l$	live form,	where l is a special constant
$N \rightarrow c(N_1, ..., N_k)$	constructor form,	where c is from a set of constructors and can also have arity 0

and the following extended forms:

$N' \rightarrow N$	copy form
$N' \rightarrow c_i^{-1}(N)$	selector form
$N' \rightarrow [N]R'$	conditional form,

where R' is of forms l, $c(N_1, ..., N_k)$, and N''. Symbols d, l, and c's are terminals, and symbols N, $N_1, ..., N_k$, N', N'' are nonterminals. The extended forms are simplified away using the algorithm below, where R is of forms l and $c(N_1, ..., N_k)$, which are called good forms.

[1]The presentation here includes minor notational changes and simplifications. In particular, in [29], the condition in the first production for a binding expression is unnecessary.

input: productions P of standard forms and extended forms;
repeat
  if P contains $N' \to N$ and $N \to R$, add $N' \to R$ to P;
  if P contains $N' \to c_i^{-1}(N)$ and $N \to l$, add $N' \to l$ to P;
  if P contains $N' \to c_i^{-1}(N)$ and $N \to c(N_1, ..., N_k)$, add $N' \to N_i$ to P;
  if P contains $N' \to [N]R'$ and $N \to R$, add $N' \to R'$ to P;
until no more productions can be added;
output: the resulting productions in P that are of good forms.

Throughout the paper, we use R' to denote right-side forms l, $c(N_1, ..., N_k)$, and N''. We use R to denote right-side good forms l and $c(N_1, ..., N_k)$; when R is a variable whose value could be an N form, it is accompanied by a test to ensure that its value is a good form.

In the application, extended forms are constructed from programs: for each program construct below on the left, the corresponding productions on the right are constructed, where a nonterminal associated with (at the left upper corner of) a program point denotes the liveness pattern for the values at that point.

function definition:	
$f(^{N_1:}v_1, ..., ^{N_n:}v_n) \triangleq e$	$N_i \to N_i'$ for $i = 1..n$ and for each occurrence of $^{N_i':}v_i$ in e
data construction:	
$^{N:}c(^{N_1:}e_1, ..., ^{N_n:}e_n)$	$N_i \to c_i^{-1}(N)$ for $i = 1..n$
selector application:	
$^{N:}c_i^{-1}(^{N_1:}e)$	$N_1 \to [N]c(\overbrace{d, ..., d}^{i-1}, N, \overbrace{d, ..., d}^{n-i})$ for c of arity n
tester application:	
$^{N:}c?(^{N_1:}e)$	$N_1 \to [N]c(\overbrace{d, ..., d}^{n})$ for each possible c of arity n
primitive operation:	
$^{N:}p(^{N_1:}e_1, ..., ^{N_n:}e_n)$	$N_i \to [N]l$ for $i = 1..n$
conditional:	
$^{N:}\textbf{if}\ ^{N_1:}e_1\ \textbf{then}\ ^{N_2:}e_2\ \textbf{else}\ ^{N_3:}e_3$	$N_1 \to [N]l,\ N_2 \to N,\ N_3 \to N$
binding:	
$^{N:}\textbf{let}\ u = {}^{N_1:}e_1\ \textbf{in}\ ^{N_2:}e_2$	$N_1 \to N_1'$ for each free occurrence of $^{N_1':}u$ in e_2, $N_2 \to N$
function application:	
$^{N:}f(^{N_1:}e_1, ..., ^{N_n:}e_n)$ where $f(^{N_1':}v_1, ..., ^{N_n':}v_n) = {}^{N':}e$	$N_i \to [N]N_i'$ for $i = 1..n$, $N' \to N$

Standard forms are given in user queries to indicate program points of interest and liveness patterns of interest at those points. For example, a user query $N \to l$ indicates that the entire value at point N is of interest. Simplification aims to add standard forms that capture the effects of extended forms. After simplification, program points whose associated nonterminals do not have a right-side good form are identified as dead.

All the production forms here are the same as or similar to those studied by many people. For example, standard forms are as in [17, 25, 11], copy forms are common in grammars, selector forms are first seen in [25], and conditional forms have counterparts in [4, 20]. Overall, the constraints and simplifications rules here extend those by Jones and Muchnick [25].

Notation. We use a set-based language. It is based on SETL [55, 56] extended with a fixed-point operation by Cai and Paige [10]; we allow sets of heterogeneous elements and extend the language with pattern matching.

Primitive data types are sets, tuples, and maps, i.e., binary relations represented as sets of 2-tuples. Their syntax and operations on them are summarized below:

$\{X_1, ..., X_n\}$	a set with elements X_1,...,X_n
$[X_1, ..., X_n]$	a tuple with elements X_1,...,X_n in order
$\{[X_1, Y_1], ..., [X_n, Y_n]\}$	a map that maps X_1 to Y_1, ..., X_n to Y_n
$\{\}$	empty set
$S \cup T,\ S - T$	union and difference, respectively, of sets S and T
S **with** X, S **less** X	$S \cup \{X\}$ and $S - \{X\}$, respectively
$S \subseteq T$	whether S is a subset of T
X **in** S, X **notin** S	whether or not, respectively, X is an element of S
$\#S$	number of elements in set S
$T(I)$	I'th component of tuple T
dom M	domain of map M, i.e., $\{X : [X, Y]$ **in** $M\}$
$M\{X\}$	image set of X under map M, i.e., $\{Y : [Z, Y]$ **in** $M \mid Z = X\}$
inv M	inverse of map M, i.e., $\{[Y, X] : [X, Y]$ **in** $M\}$

We use the notation below for pattern matching against constants and tuples. The first returns true if X is constant c and false otherwise. The second returns false if X is not a tuple of length n; otherwise, it binds Y_i to the ith component of X if Y_i is an unbound variable, and otherwise, recursively tests whether the ith component of X matches Y_i, until either a test fails or all unbound variables in the pattern become bound.

X **of** c, where c is a constant	whether X is constant c
X **of** $[Y_1, ..., Y_n]$	whether X matches pattern $[Y_1, ..., Y_n]$

We use the notation below for set comprehension, where X and Z are expressions that use Y_i's. Y_i's enumerate elements of all S_i's; for each combination of $Y_1, ..., Y_n$, if the Boolean value of expression Z is true, then the value of expression X forms an element of the resulting set. Each Y_i can be a tuple, in which case an enumerated element of S_i is first matched against it.

$\{X : Y_1$ **in** $S_1, ..., Y_n$ **in** $S_n \mid Z\}$	set former
$\{X : Y_1$ **in** $S_1, ..., Y_n$ **in** $S_n\}$	abbreviation of $\{X : Y_1$ **in** $S_1, ..., Y_n$ **in** $S_n \mid true\}$
$\{Y$ **in** $S \mid Z\}$	abbreviation of $\{Y : Y$ **in** $S \mid Z\}$

We use the following least-fixed-point operation to denote the minimum element Y, with respect to partial ordering $\subseteq$, that satisfies the condition $X \subseteq Y$ and $F(Y) = Y$:

$$\mathbf{LFP}_{X,\ \subseteq}\ (F(Y), Y) \qquad \text{least-fixed-point operation}$$

We use standard control constructs **while**, **for**, **if**, and **case**, and we use indentation to indicate scoping. We abbreviate $X := X$ **op** Y as X **op** $:= Y$. Also, we abbreviate $X_1 := Y; ...; X_n := Y$ as $X_1, ..., X_n := Y$.

A set-based fixed-point specification. A straightforward translation of the problem into a fixed-point specification using sets and tuples is as follows. First, represent the right-side R' forms as follows:

$$\begin{array}{lll} l & \text{as} & l, \text{ where } l \text{ is a special constant} \\ c(N_1, ..., N_k) & \text{as} & [c, [N_1, ..., N_k]] \\ N & \text{as} & N \end{array} \tag{1}$$

and represent the productions as follows:

$$\begin{array}{lll} N' \to R' & \text{as} & [N', \text{representation of } R'] \\ N' \to c_i^{-1}(N) & \text{as} & [N', c, i, N] \\ N' \to [N]R' & \text{as} & [N', N, \text{representation of } R'] \end{array} \tag{2}$$

This representation allows us to distinguish all the production forms by simple pattern matching against constants and tuples of different lengths. The three kinds of productions in (2) happen to correspond to tuples of different lengths. The three kinds of right-side R' forms in (1) correspond to special constant l, pair, and other; we also need to tell the R form from the N form, so for convenience, we use the following two predicates:

$$\begin{array}{lll} R'\ isR & = & R' \textbf{ of } l \textbf{ or } R' \textbf{ of } [C, T] \\ R'\ isN & = & \textbf{not } (R' \textbf{ of } l \textbf{ or } R' \textbf{ of } [C, T]) \end{array} \tag{3}$$

We could use tags to represent sets with heterogeneous elements [42], but tags are not needed here.

The algorithm can be specified as follows. The input is a set P of productions in the new representation. The loop computes the minimum set Q that satisfies $P \subseteq Q$ and $F(Q) \subseteq Q$, where $F(Q)$ captures, line-by-line, the four rules in the loop body:

$$\begin{array}{rl} F(Q) = & \{[N',R] : [N',N] \textbf{ in } Q,\ [N,R] \textbf{ in } Q \mid R\ isR\} \cup \\ & \{[N',l] : [N',C,I,N] \textbf{ in } Q,\ [N,l] \textbf{ in } Q\} \cup \\ & \{[N',T(I)] : [N',C,I,N] \textbf{ in } Q, [N,[C,T]] \textbf{ in } Q\} \cup \\ & \{[N',R'] : [N',N,R'] \textbf{ in } Q,\ [N,R] \textbf{ in } Q \mid R\ isR\} \end{array} \tag{4}$$

Since $F(Q) \subseteq Q$ iff $F(Q) \cup Q = Q$, the loop computes the minimum set Q that satisfies $P \subseteq Q$ and $F(Q) \cup Q = Q$, denoted as the least fixed point:

$$\mathbf{LFP}_{P, \subseteq} (F(Q) \cup Q,\ Q) \tag{5}$$

The output is the set O of resulting productions whose right side is a good form:

$$O = \{[N,R] \textbf{ in } \mathbf{LFP}_{P, \subseteq} (F(Q) \cup Q,\ Q) \mid R\ isR\} \tag{6}$$

The representation of constraints using SETL tuples is immaterial to the problem. However, efficient algorithms for simplifying the constraints require the use of auxiliary maps, as discussed in Section 4; both for discovering such auxiliary expressions and for representing and manipulating them at a high level, uniform notation helps in the precise presentation of the systematic design method.

3 Approach

The method has three steps: (1) dominated convergence, (2) finite differencing, and (3) real-time simulation.

Dominated convergence [10] transforms a set-based fixed-point specification into a **while**-loop. The idea is to perform a small update operation in each iteration. The fixed-point expression $\mathbf{LFP}_{P, \subseteq} (F(Q) \cup Q,\ Q)$ in (6) is transformed into the following **while**-loop:

$$\begin{array}{l} Q := P; \\ \textbf{while exists } p \textbf{ in } F(Q) - Q \\ \quad Q \textbf{ with} := p; \end{array} \tag{7}$$

This code is followed by

$$O = \{[N,R] \textbf{ in } Q \mid R\ isR\}; \tag{8}$$

Finite differencing [41, 39] transforms expensive set operations in a loop into incremental operations. The idea is to compute expensive expressions, say exp_1 through exp_n, in the loop, say called $LOOP$, by maintaining the invariants $E_1 = exp_1$ through $E_n = exp_n$, for fresh variables E_1 through E_n, incrementally with the execution of the loop. We denote the transformed loop as

$$\Delta E_1, ..., E_n \ \langle\ LOOP\ \rangle$$

which associates with each assignment in $LOOP$ appropriate incremental updates to E_1 through E_n, and replaces the expensive computations of exp_1 through exp_n in $LOOP$ with E_1 through E_n, respectively. For our program (7) and (8) from Step 1, expensive expressions are the one that computes O and others that are needed for computing $F(Q) - Q$. They are initialized together with the assignment $Q := P$ and computed incrementally as Q is augmented by p in each iteration.

Real-time simulation [40, 9] selects appropriate data structures for representing sets so that operations on them can be implemented efficiently. The idea is to design sophisticated linked structures based on how sets and set elements are accessed, so that each operation can be performed in constant time with at most a constant (a small fraction) factor of overall space overhead.

For our program, in Step 2, initialization of expensive set expressions is done by adding each production in P into Q one at a time, yielding incremental computation of these expressions similar to that in the loop body. This results in large initialization code.

To overcome this problem, Cai and Paige [10] propose transformations of fixed-point expressions in Step 1 so that much simpler initialization code can be obtained by the subsequent finite differencing. However, those transformations place code for handling different cases of input data all in the loop body and make the loop body larger and the complexity analysis less accurate. We propose more general transformations that simplify the initialization code without complicating the loop body; in fact, they make the overall code even simpler and make the overall complexity analysis accurate.

For Step 3, to support associative access, i.e., locating a given value in a set, say X **in** S, in constant time, Paige [40] proposes to use based representation. The basic idea is to represent X as a record with a field indicating whether it is in S. This works only if there are a constant number of sets like S. However, our application involves a linear number of sets like S. We discuss data structure choices—arrays, linked lists, and hash tables—and the trade-offs. Our complexity analysis suggests that simple linked lists, called unbased representation, work well in our application, and our experiments support this.

4 Finite differencing

We perform finite differencing on (7). This consists of the following steps: identifying expensive subexpressions, discovering auxiliary expressions, finite differencing of the loop body, handling initialization, and eliminating dead code.

Identifying expensive subexpressions. The output O in (8) and expensive subexpressions used to compute O need to be computed incrementally in the loop. The latter expressions are $E1$ to $E4$, one for each of the sets in $F(Q)$ in (4), and W, the workset:

$$\begin{aligned}
E1 &= \{[N',R] : [N',N] \textbf{ in } Q, [N,R] \textbf{ in } Q \mid R\ isR\} \\
E2 &= \{[N',l] : [N',C,I,N] \textbf{ in } Q, [N,l] \textbf{ in } Q\} \\
E3 &= \{[N',T(I)] : [N',C,I,N] \textbf{ in } Q, [N,[C,T]] \textbf{ in } Q\} \\
E4 &= \{[N',R'] : [N',N,R'] \textbf{ in } Q, [N,R] \textbf{ in } Q \mid R\ isR\} \\
W &= F(Q) - Q = E1 \cup E2 \cup E3 \cup E4 - Q
\end{aligned} \tag{9}$$

Thus, the overall computation becomes

$$\Delta O, E1, E2, E3, E4, W \ \langle \ \begin{array}{l} Q := P; \\ \textbf{while exists } p \textbf{ in } W \\ \quad Q \textbf{ with} := p; \ \rangle \end{array} \tag{10}$$

Discovering auxiliary expressions. To compute $E1$ to $E4$ incrementally with respect to Q **with** $:= p$, the following auxiliary expressions $E11$ to $E41$ are maintained. Expression $E11$ maps N to N' if there is a production of form $N' \rightarrow N$. Expression $E21$ maps N to N' and expression $E31$ maps $[c,N]$ to $[N',i]$ if there is a production of the form $N' \rightarrow c_i^{-1}(N)$. Expression $E41$ maps N to $[N',R']$ if there is a production of form $N' \rightarrow [N]R'$.

$$\begin{aligned}
E11 &= \{[N,N'] : [N',N] \textbf{ in } Q \mid N\ isN\} \\
E21 &= \{[N,N'] : [N',C,I,N] \textbf{ in } Q\} \\
E31 &= \{[[C,N],[N',I]] : [N',C,I,N] \textbf{ in } Q\} \\
E41 &= \{[N,[N',R']] : [N',N,R'] \textbf{ in } Q\}
\end{aligned} \tag{11}$$

These expressions are introduced for differentiating $E1$ to $E4$, respectively. For example, $E11$ is introduced for differentiating $E1$ in (9) after adding an element $[N,R]$ in Q—we need to add $[N',R]$ to $E1$ for all $[N',N]$ in Q, i.e., for all N' in $E11\{N\}$. These expressions can be obtained systematically based on the set formers in (9): after adding an element corresponding to one enumerator, create based on the other enumerator a map from variables that are already bound to variables yet unbound. For example, consider $E3$ and adding an element $[N,[C,T]]$ in Q. Then, for $[N',C,I,N]$ in Q, variables C and N are bound, and N' and I are not. So, we create a map from $[C,N]$ to $[N',I]$ for each $[N',C,I,N]$ in Q, which is $E31$. Now, the overall computation becomes

$$\Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \; \langle \quad \begin{array}{l} Q := P; \\ \textbf{while exists } p \textbf{ in } W \\ \quad Q \textbf{ with} := p; \; \rangle \end{array} \tag{12}$$

These auxiliary maps provide, at a high level, the indexing needed to support efficient incremental updates.

Transforming loop body. We apply finite differencing to the loop body. This means that we differentiate O, $E1$ to $E4$, W, and $E11$ to $E41$ with respect to Q **with** $:= p$ in (12):

$$\Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \; \langle \; Q \textbf{ with} := p; \; \rangle \tag{13}$$

Based on the elements added to W, which is through $E1$ to $E4$, p can be of forms $[N,l]$, $[N,[C,T]]$, and $[N',N]$ where $N\ isN$. For each form of p, we list how the sets O, $E1$ to $E4$, and $E11$ to $E41$ are updated; those not listed do not change:

$$\begin{array}{ll}
\text{if } p \text{ is of form } [N,l], & O \textbf{ with} := [N,l]; \\
 & E1 \cup := \{[N',l] : N' \textbf{ in } E11\{N\}\}; \\
 & E2 \cup := \{[N',l] : N' \textbf{ in } E21\{N\}\}; \\
 & E4 \cup := \{[N',R'] \textbf{ in } E41\{N\}\}; \\
\text{if } p \text{ is of form } [N,[C,T]], & O \textbf{ with} := [N,[C,T]]; \\
 & E1 \cup := \{[N',[C,T]] : N' \textbf{ in } E11\{N\}\}; \\
 & E3 \cup := \{[N',T(I)] : [N',I] \textbf{ in } E31\{[C,N]\}\}; \\
 & \text{same update to } E4 \text{ as above} \\
\text{if } p \text{ is of form } [N',N], \text{ where } N\ isN, & E1 \cup := \{[N',R] : R \textbf{ in } O\{N\}\}; \\
 & E11 \textbf{ with} := [N,N'];
\end{array}$$

Also, for each of the forms, we do two things to update W. First, with anything added into $E1$ to $E4$, if it is not in Q, then it is added to W. Second, remove p from W.

We can group the first two forms of p for their updates to $E1$ and $E4$, group the removal of p from W for all forms at the beginning, and put all these updates after the original assignment Q **with** $:= p$, yielding the following complete code for the loop body:

$$\begin{array}{l}
Q \textbf{ with} := p; \\
W \textbf{ less} := p; \\
\textbf{case } p \textbf{ of} \\
\quad [N,R], \text{ where } R\ isR : \\
\quad\quad O \textbf{ with} := [N,R]; \\
\quad\quad E1 \cup := \{[N',R] : N' \textbf{ in } E11\{N\}\}; \\
\quad\quad W \cup := \{[N',R] : N' \textbf{ in } E11\{N\} \mid [N',R] \textbf{ notin } Q\}; \\
\quad\quad E4 \cup := \{[N',R'] \textbf{ in } E41\{N\}\}; \\
\quad\quad W \cup := \{[N',R'] \textbf{ in } E41\{N\} \mid [N',R'] \textbf{ notin } Q\}; \\
\quad [N,l] : \\
\quad\quad E2 \cup := \{[N',l] : N' \textbf{ in } E21\{N\}\}; \\
\quad\quad W \cup := \{[N',l] : N' \textbf{ in } E21\{N\} \mid [N',l] \textbf{ notin } Q\}; \\
\quad [N,[C,T]] : \\
\quad\quad E3 \cup := \{[N',T(I)] : [N',I] \textbf{ in } E31\{[C,N]\}\}; \\
\quad\quad W \cup := \{[N',T(I)] : [N',I] \textbf{ in } E31\{[C,N]\} \mid [N',T(I)] \textbf{ notin } Q\}; \\
\quad [N',N], \text{ where } N\ isN : \\
\quad\quad E1 \cup := \{[N',R] : R \textbf{ in } O\{N\}\}; \\
\quad\quad W \cup := \{[N',R] : R \textbf{ in } O\{N\} \mid [N',R] \textbf{ notin } Q\}; \\
\quad\quad E11 \textbf{ with} := [N,N'];
\end{array} \tag{14}$$

These updates are keys for achieving high efficiency: after adding a new production, we consider only productions that are directly affected. This makes the analysis proceed in an incremental fashion.

Initialization. Sets O, $E1$ to $E4$, W, and $E11$ to $E41$ need to be initialized together with $Q := P$ in (12). To do this, we add each p from P into Q one by one, also incrementally:

$$\begin{array}{l} \Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \ \langle\ Q := \{\};\ \rangle \\ \textbf{for}\ p\ \textbf{in}\ P \\ \quad \Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \ \langle\ Q\ \textbf{with} := p;\ \rangle \end{array} \tag{15}$$

The first line becomes

$$O, E1, E2, E3, E4, W, E11, E21, E31, E41, Q := \{\};$$

and, in the body of the **for** loop, which is the same as (13), we have the same four cases of p as in the loop body (14) and the following two additional forms of p, for which we list how the sets O, $E1$ to $E4$, and $E11$ to $E41$ are updated:

$$\begin{array}{ll} \text{if } p \text{ is of form } [N', C, I, N], & E2 \cup := \{[N', l] : l\ \textbf{in}\ Q\{N\}\}; \\ & E21\ \textbf{with} := [N, N']; \\ & E3 \cup := \{[N', T(I)] : [C, T]\ \textbf{in}\ Q\{N\}\}; \\ & E31\ \textbf{with} := [[C, N], [N', I]]; \\ \text{if } p \text{ is of form } [N', N, R'], & E4 \cup := \{[N', R'] : R\ \textbf{in}\ Q\{N\} \mid R\ isR\}; \\ & E41\ \textbf{with} := [N, [N', R']]; \end{array}$$

Also, for each of the forms, W is handled as in the loop body. We obtain the following complete code for initialization:

$$\begin{array}{l} O, E1, E2, E3, E4, W, E11, E21, E31, E41, Q := \{\}; \\ \textbf{for}\ p\ \textbf{in}\ P \\ \quad Q\ \textbf{with} := p; \\ \quad W\ \textbf{less} := p; \\ \quad \textbf{case}\ p\ \textbf{of} \\ \quad\quad \text{same four cases of } p \text{ as in the loop body} \\ \quad\quad [N', C, I, N] : \\ \quad\quad\quad E2\ \ \cup := \{[N', l] : l\ \textbf{in}\ Q\{N\}\}; \\ \quad\quad\quad W\ \ \cup := \{[N', l] : l\ \textbf{in}\ Q\{N\} \mid [N', l]\ \textbf{notin}\ Q\}; \\ \quad\quad\quad E21\ \textbf{with} := [N, N']; \\ \quad\quad\quad E3\ \ \cup := \{[N', T(I)] : [C, T]\ \textbf{in}\ Q\{N\}\}; \\ \quad\quad\quad W\ \ \cup := \{[N', T(I)] : [C, T]\ \textbf{in}\ Q\{N\} \mid [N', T(I)]\ \textbf{notin}\ Q\}; \\ \quad\quad\quad E31\ \textbf{with} := [[C, N], [N', I]]; \\ \quad\quad [N', N, R'] : \\ \quad\quad\quad E4\ \ \cup := \{[N', R'] : R\ \textbf{in}\ Q\{N\} \mid R\ isR\}; \\ \quad\quad\quad W\ \ \cup := \{[N', R'] : R\ \textbf{in}\ Q\{N\} \mid R\ isR, [N', R']\ \textbf{notin}\ Q\}; \\ \quad\quad\quad E41\ \textbf{with} := [N, [N', R']]; \end{array} \tag{16}$$

Dead-code elimination. Since only O is the desired output, it is easy to see that $E1$ to $E4$ are not needed, i.e., they are dead. Furthermore, Q can be eliminated using the equivalences:

$$\begin{array}{rcl} [N, R]\ \textbf{in}\ Q, \text{ where } R\ isR & \iff & [N, R]\ \textbf{in}\ O \\ [N', N]\ \textbf{in}\ Q, \text{ where } N\ isN & \iff & [N, N']\ \textbf{in}\ E11 \end{array}$$

We obtain the following complete algorithm:

$$
\begin{array}{l}
O, W, E11, E21, E31, E41 := \{\};\\
\textbf{for } p \textbf{ in } P\\
\quad W \textbf{ less} := p;\\
\quad \textbf{case } p \textbf{ of}\\
\quad\quad \text{same four cases of } p \text{ as in the loop body}\\
\quad\quad [N', C, I, N] :\\
\quad\quad\quad W \cup := \{[N', l] : l \textbf{ in } O\{N\} \mid [N', l] \textbf{ notin } O\};\\
\quad\quad\quad E21 \textbf{ with} := [N, N'];\\
\quad\quad\quad W \cup := \{[N', T(I)] : [C, T] \textbf{ in } O\{N\} \mid [T(I), N'] \textbf{ notin } E11\};\\
\quad\quad\quad E31 \textbf{ with} := [[C, N], [N', I]];\\
\quad\quad [N', N, R'] :\\
\quad\quad\quad W \cup := \{[N', R'] : R \textbf{ in } O\{N\} \mid \textbf{if } R' \ isR \textbf{ then } [N', R'] \textbf{ notin } O \textbf{ else } [R', N'] \textbf{ notin } E11\};\\
\quad\quad\quad E41 \textbf{ with} := [N, [N', R']];\\
\textbf{while exists } p \textbf{ in } W\\
\quad W \textbf{ less} := p;\\
\quad \textbf{case } p \textbf{ of}\\
\quad\quad [N, R], \text{ where } R \ isR :\\
\quad\quad\quad O \textbf{ with} := [N, R];\\
\quad\quad\quad W \cup := \{[N', R] : N' \textbf{ in } E11\{N\} \mid [N', R] \textbf{ notin } O\};\\
\quad\quad\quad W \cup := \{[N', R'] \textbf{ in } E41\{N\} \mid \textbf{if } R' \ isR \textbf{ then } [N', R'] \textbf{ notin } O \textbf{ else } [R', N'] \textbf{ notin } E11\};\\
\quad\quad [N, l] :\\
\quad\quad\quad W \cup := \{[N', l] : N' \textbf{ in } E21\{N\} \mid [N', l] \textbf{ notin } O\};\\
\quad\quad [N, [C, T]] :\\
\quad\quad\quad W \cup := \{[N', T(I)] : [N', I] \textbf{ in } E31\{[C, N]\} \mid [T(I), N'] \textbf{ notin } E11\};\\
\quad\quad [N', N], \text{ where } N \ isN :\\
\quad\quad\quad W \cup := \{[N', R] : R \textbf{ in } O\{N\} \mid [N', R] \textbf{ notin } O\};\\
\quad\quad\quad E11 \textbf{ with} := [N, N'];
\end{array}
\tag{17}
$$

where $W \cup := \{X : Y \textbf{ in } S \mid Z\}$ is implemented as

$$
\begin{array}{l}
\textbf{for } Y \textbf{ in } S\\
\quad \textbf{if } Z \textbf{ then}\\
\quad\quad W \textbf{ with} := X;
\end{array}
\tag{18}
$$

Complexity analysis. We don't know the data structure that implements sets O, W, and $E11$ to $E41$ yet, but for now we assume that set initialization $S := \{\}$, retrieval of an arbitrary element in a set by **for** or **while** or an indexed element by $T(I)$, element addition and deletion S **with/less** X, and associative access X **notin** S and $M\{X\}$ each takes $\mathcal{O}(1)$ time. Other operations, such as pattern matching, clearly take $\mathcal{O}(1)$ time.

Besides input size $\#P$ and output size $\#O$, i.e., the number of productions in input and output, respectively, we use the following parameters. The meanings of these parameters are based on how the constraints were constructed. Note that sets $E11$ to $E41$ only grow during the computation, so we consider their values at the end.

- Let a be the maximum of $\#E21\{N\}$, $\#E31\{[C,N]\}$, and $\#E41\{N\}$ for any N and C. Note that $E21$ and $E31$ depend only on selector forms, and $E41$ on conditional forms, in the input.

  Meaning: In the application, these forms are built from programs, and a is the maximum of the arities of constructors, primitive functions, and user-defined functions and the number of possible outermost constructors in the argument of a tester (such as $null$). In fact, $\#E21\{N\}$ and $\#E31\{[C,N]\}$ are bounded by the maximum arity of constructors only.

- Let h be the maximum number of nonterminals to the left of a nonterminal:

$$h = \max{}_{N \textbf{ in dom } E11} \#E11\{N\} \tag{19}$$

  Meaning: In the application, for productions built from programs, $\#E11\{N\} \leq 2$ for any N (2 for a conditional expression, 1 for a binding expression and a function call, 0 for others). However, $E11$ and

h may grow during simplification, by addition to $E3$, through selecting a component of a construction, or to $E4$, through enabling a condition. In fact, from enabling conditions at function calls, h is at least as large as the maximum number of live call sites of any function, but selecting components of constructions may yield even larger h.

- Let g be the maximum number of good forms a nonterminal goes to:

$$g = \max_{N \textbf{ in dom } O} \#O\{N\} \tag{20}$$

Meaning: In the application, a good form is either l or the right side of a constructor form constructed at the argument of a selector or a tester, and testers together generate no more than a constructor forms. Thus, g corresponds to the maximum of a and the maximum number of selector applications into whose arguments the value constructed at a program point might flow.

- Let r be the size of the domain of O:

$$r = \#\textbf{dom } O \tag{21}$$

Meaning: In the application, r is the number of live program points. Note that $\#O \leq r * g$. If g is a constant, then $\#O$ is $\mathcal{O}(r)$.

- Let n be the number of nonterminals in P.

Meaning: In the application, n is the number of program points plus the number of nonterminals introduced in a user query. A user query usually has a small number of productions, and at most $a+1$ productions are constructed at each program point, so usually $\#P \leq n * a$. If a is a constant, then $\#P$ is $\mathcal{O}(n)$.

Parameter n is not needed in the complexity analysis, but it best captures program size. Also, n bounds h, and $\#P$ bounds g; the latter is because all good forms are in the given productions, so there are at most $\#P$ of them.

The complexity is the sum of (i) a constant for each element considered for addition to W, as in all the assignments to W, (ii) a constant for each element in W, as in the iterations, and (iii) a constant for each element in P, as in the initialization. Clearly, (ii) is bounded by (i), and (iii) is $\mathcal{O}(\#P)$. The total for (i) is the sum of (c1) to (c8) below, where (c1) to (c5) are for cases 1 to 4 in both the iteration and initialization, and (c6) to (c8) are for cases 5 and 6 in the initialization, explained below.

cases 1-3:	$\Sigma_{[N,R] \textbf{ in } O} \#E11\{N\}$	(c1)
	$\Sigma_{[N,l] \textbf{ in } O} \#E21\{N\}$	(c2)
	$\Sigma_{[N,[C,T]] \textbf{ in } O} \#E31\{[C,N]\}$	(c3)
	$\Sigma_{[N,R] \textbf{ in } O} \#E41\{N\}$	(c4)
case 4:	$\Sigma_{[N,N'] \textbf{ in } E11} \#O\{N\}$	(c5)
case 5:	$\Sigma_{[N',C,I,N] \textbf{ in } P} \#\{l \textbf{ in } O\{N\}\}$	(c6)
	$\Sigma_{[N',C,I,N] \textbf{ in } P} \#\{[C,T] \textbf{ in } O\{N\}\}$	(c7)
case 6:	$\Sigma_{[N',N,R'] \textbf{ in } P} \#\{R \textbf{ in } O\{N\}\}$	(c8)

For each p of form $[N,R]$, all N' **in** $E11\{N\}$ and all $[N',R']$ **in** $E41\{N\}$ are considered; since each p of form $[N,R]$ is added to set O, the total complexity for case 1 is (c1) plus (c4). For each p of form $[N,l]$, we also add (c2), and for each p of form $[N,[C,T]]$, we also add (c3). Similarly, for each p of form $[N',N]$, we have (c5). For cases 5 and 6 in the initialization, we have (c6) to (c8).

Using the parameters introduced above, we have

$$\begin{aligned} &(\text{c1}) \leq h * \#O \\ &(\text{c2}) \leq a * r \\ &(\text{c3}) \leq a * \#O \\ &(\text{c4}) \leq a * \#O \end{aligned} \tag{22}$$

Note that

$$(c1) = (c5) = \Sigma_{N \textbf{ in dom } O} \#E11\{N\} * \#O\{N\} \tag{23}$$

A second way of estimating (c1) and (c5) is

$$\begin{array}{rll} (c1) = (c5) & \leq \#\{[N,N'] \textbf{ in } E11 \mid N \textbf{ in dom } O\} * g & \text{by (23)} \\ & = \#\{[N',N] \textbf{ in } Q \mid N \textbf{ in dom } O\} * g & \text{by definition of } E11 \\ & \leq (\#\{[N',N] \textbf{ in } P \mid N \textbf{ in dom } O\} + & \text{those of form } [N',N] \text{ in } P \\ & \quad \#\{[N',N] \textbf{ in } E3 \mid N \textbf{ in dom } O\} + & \text{those of form } [N',N] \text{ in } E3 \\ & \quad \#\{[N',N] \textbf{ in } E4 \mid N \textbf{ in dom } O\}) * g & \text{those of form } [N',N] \text{ in } E4 \text{ where } N \ isN \\ & & \text{these three contribute all of form } [N',N] \text{ in } Q \\ & \leq (r + (c3) + (c4)) * g & \\ & \leq (r + a * \#O + a * \#O) * g & \end{array} \tag{24}$$

Therefore, (c1) and (c5) are $\mathcal{O}(\#O * g * a)$. Thus, the sum of (c1) through (c5) is $\mathcal{O}(\#O * (h + a))$, using the first way of estimating (c1) and (c5), and $\mathcal{O}(\#O * g * a)$, using the second way. Also,

$$\text{(c6), (c7), (c8)} \leq g * \#P \tag{25}$$

Thus, the total complexity of (i) to (iii) is $\mathcal{O}(\#O * \min(h + a, g * a) + \#P * g + \#P)$, which is

$$\mathcal{O}(\#O * \min(h + a, g * a) + \#P * g) \tag{26}$$

since $\#O \neq 0$ and thus $g \neq 0$ in the application.

In the application, productions in P with right sides in good forms are from the user query, and they are put in O in the initialization; if we assume there is a constant number of them, i.e., $\#O$ and thus g is a constant during initialization, then (c6) to (c8) are $\mathcal{O}(\#P)$, and the total complexity is $\mathcal{O}(\#O*\min(h+a, g*a)+\#P)$. Section 5 proposes a high-level transformation that allows us to both avoid the code duplication in algorithm (17) and give the complexity $\mathcal{O}(\#O * \min(h+a, g*a) + \#P)$ without this assumption about the user query.

5 Higher-level design and analysis

Avoiding duplication of code for initialization. Algorithm (17) duplicates the code in the loop body in the initialization. Cai and Paige [10] proposed a high-level transformation that can drastically simplify the initialization and do all the work in the loop body. By Theorem 5 in [10], the fixed-point expression (5) is equivalent to

$$\textbf{LFP}_{\{\},\subseteq}(P \cup F(Q) \cup Q,\, Q) \tag{27}$$

which can be transformed into

$$\begin{array}{l} Q := \{\}; \\ \textbf{while exists } p \textbf{ in } P \cup F(Q) - Q \\ \quad Q \textbf{ with} := p; \end{array} \tag{28}$$

This merges the initialization for $Q := P$ into the iteration and thus avoids code duplication. After finite differencing, we obtain the following complete algorithm:

$$\begin{array}{l} O, E11, E21, E31, E41 := \{\}; \\ W := P; \\ \text{same iteration as in algorithm (17) except with} \\ \quad \text{cases 5 and 6 in the initialization appended to the loop body} \end{array} \tag{29}$$

However, this merging reduces the accuracy of the complexity analysis. The complexity of the resulting program is analyzed as in Section 4, except that there is no (iii) here, but (ii) here equals the sum of (ii) and (iii) there. So, the total complexity is again $\mathcal{O}(\#O * \min(h + a, g * a) + \#P * g)$. We can not obtain $\mathcal{O}(\#O * \min(h+a, g*a) + \#P)$ here, even if we have the additional assumption about the user query, as we did at the end of Section 4, because (c6) to (c8) are now from the main loop, where g is not bounded by a constant.

Simplifying overall code and making complexity analysis accurate. We propose a general method that not only eliminates code duplication completely but also yields overall even smaller code and more accurate complexity. The method is to merge into the main loop only the cases in the initialization that must be handled in the main loop, not the cases that are needed only in initialization. This is good for two reasons: performing work under the same cases together allows its total amount to be counted more accurately, and separating work under different cases allows each to be counted more accurately. Our method is supported by the following theorem. It generalizes the transformation from (5) to (27).

Theorem 5.1 *For all* $P_0 \subseteq P$, $\mathbf{LFP}_{P_0,\subseteq}((P-P_0)\cup F(Q)\cup Q, Q)$ *exists if and only if* $\mathbf{LFP}_{P,\subseteq}(F(Q)\cup Q, Q)$ *exists, and if they exist, they are equal.*

Proof: Define predicates C_0 and C_1 by

$$\begin{array}{lcl} C_0(Q) & = & P_0 \subseteq Q \land Q = (P-P_0)\cup F(Q)\cup Q \\ C_1(Q) & = & P \subseteq Q \land Q = F(Q)\cup Q \end{array}$$

Let S_i contain the sets satisfying C_i. $\mathbf{LFP}_{P_0,\subseteq}((P-P_0)\cup F(Q)\cup Q, Q)$ exists iff S_0 is non-empty and has a least element; if such an element exists, it is the least fixed point. Similarly, $\mathbf{LFP}_{P,\subseteq}(F(Q)\cup Q, Q)$ exists iff S_1 is non-empty and has a least element; if such an element exists, it is the least fixed point. Thus, it suffices to show $S_0 = S_1$, or equivalently, $C_0(Q) \Leftrightarrow C_1(Q)$.

First, we show $C_0(Q) \Rightarrow C_1(Q)$. Suppose Q satisfies C_0. By definition of C_0, $P_0 \subseteq Q$, so $P \subseteq (P-P_0)\cup Q$. This and $Q = (P-P_0)\cup F(Q)\cup Q$ imply $P \subseteq Q$, which implies $Q = (P-P_0)\cup Q$. Using this equality, $Q = (P-P_0)\cup F(Q)\cup Q$ simplifies to $Q = F(Q)\cup Q$. Thus, Q satisfies C_1.

Second, we show $C_1(Q) \Rightarrow C_0(Q)$. Suppose Q satisfies C_1. By definition of C_1, $P \subseteq Q$. This and the hypothesis $P_0 \subseteq P$ imply $P_0 \subseteq Q$. Also, $P \subseteq Q$ implies $Q = (P-P_0)\cup Q$. Using this equality, $Q = F(Q)\cup Q$ can be transformed to $Q = (P-P_0)\cup F(Q)\cup Q$. Thus, Q satisfies C_0. QED

We apply Theorem 5.1 with

$$P_0 = \{p \textbf{ in } P \mid p \textbf{ of } [N', C, I, N] \textbf{ or } p \textbf{ of } [N', N, R']\}$$

The fixed-point expression (5) is equivalent to

$$\mathbf{LFP}_{P_0,\subseteq}(P - P_0 \cup F(Q)\cup Q,\ Q) \tag{30}$$

which is transformed into the following **while**-loop:

$$\begin{array}{l} Q := P_0; \\ \textbf{while exists } p \textbf{ in } P - P_0 \cup F(Q) - Q \\ \quad Q \textbf{ with} := p; \end{array} \tag{31}$$

This merges cases 1 through 4 in the initialization into the loop body and thus avoids code duplication. Furthermore, it makes cases 5 and 6 much simpler, because there are no good forms in P_0, hence no updates to O and thus no updates to W. After finite differencing, we obtain the following complete algorithm, which has the same iteration as in algorithm (17) and initializes O and $E11$ to $\{\}$, $E21$ through $E41$ for p in P_0 as in (17), and W to $P - P_0$:

$$\begin{array}{l} O, W, E11, E21, E31, E41 := \{\}; \\ \textbf{for } p \textbf{ in } P \\ \quad \textbf{case } p \textbf{ of} \\ \quad\quad [N', C, I, N]: \\ \quad\quad\quad E21 \textbf{ with} := [N, N']; \\ \quad\quad\quad E31 \textbf{ with} := [[C, N], [N', I]]; \\ \quad\quad [N', N, R']: \\ \quad\quad\quad E41 \textbf{ with} := [N, [N', R']]; \\ \quad\quad \textbf{other}: \\ \quad\quad\quad W \textbf{ with} := p; \\ \text{same iteration as in algorithm (17)} \end{array} \tag{32}$$

The complexity analysis is the same as in Section 4, except that the corresponding (c6) to (c8) in (i) equal zero here, and (ii) here is bounded by the sum of (ii) and (iii) there. Thus, the total complexity is

$$\mathcal{O}(\#O * \min(h + a, g * a) + \#P) \tag{33}$$

which is better than the complexity (26) obtained for (17).

Handling multiple queries. In the application, especially for interactive program manipulation environments, there can be many queries about a program. We can transform the above algorithm, so that initialization is done once in linear time in the size of the program, and simplification after each query takes time roughly linear in the number of live program points. In particular, initialization can be done concurrently with the construction of the productions.

Let P_0 be the set of productions constructed from the given program; it contains only productions of copy, selector, and conditional forms. Let P_1 be the set of productions from a user query; they are all in good forms. Thus, based on Theorem 5.1, initialization using P_0 followed by simplification using P_1 can be specified as

$$\mathbf{LFP}_{P_0,\subseteq}(P_1 \cup F(Q) \cup Q,\, Q) \tag{34}$$

which is transformed into

$$\begin{array}{l} Q := P_0; \\ \textbf{while exists } p \textbf{ in } P_1 \cup F(Q) - Q \\ \quad Q \textbf{ with} := p; \end{array} \tag{35}$$

After finite differencing, we obtain the following complete algorithm:

$$\begin{array}{l}
E11, E21, E31, E41 := \{\}; \\
\textbf{for } p \textbf{ in } P_0 \\
\quad \textbf{case } p \textbf{ of} \\
\quad\quad [N', N],\ \text{where } N\ isN: \\
\quad\quad\quad E11 \textbf{ with} := [N, N']; \\
\quad\quad [N', C, I, N]: \\
\quad\quad\quad E21 \textbf{ with} := [N, N']; \\
\quad\quad\quad E31 \textbf{ with} := [[C, N], [N', I]]; \\
\quad\quad [N', N, R']: \\
\quad\quad\quad E41 \textbf{ with} := [N, [N', R']]; \\
O, W := \{\}; \\
\textbf{for } p \textbf{ in } P_1 \\
\quad W \textbf{ with} := p; \\
\text{same iteration as in algorithm (17)}
\end{array} \tag{36}$$

Initializations of $E11$ to $E41$ depend on P_0 and are done only once. So we put them before initialization of O and W, which are updated by the iteration. Note that $E11$ may also be augmented by the iteration, so we need to preserve the value of $E11$ after the initialization. To do so, we simply use a new set $E11'$ to keep new elements of $E11$ in the iteration: set $E11' := \{\}$ immediately before the iteration, and in the iteration, replace assignments to $E11$ with assignments to $E11$', and enumerations and tests using $E11$ with those using $E11 \cup E11'$. Use of $E11'$ does not change the complexity.

Initializations of $E11$ to $E41$ clearly take $\mathcal{O}(\#P_0)$ time. The complexity of the rest is again analyzed as in Section 4, except that the corresponding (c6) to (c8) in (i) equal zero here, and (ii) here equals the sum of (ii) and (iii) there where P is replaced by P_1. Since all productions in P_1 are in good forms, $P_1 \subseteq O$. So, the total complexity for simplification after a query is

$$\mathcal{O}(\#O * \min(h + a, g * a)). \tag{37}$$

An optimization to conditional forms. For production p of form $[N, R]$ where $R\ isR$, we can add the following updates at the end of handling that form, so as to avoid unnecessarily enabling any conditional form more than once:

$$\begin{array}{ll} Q & - := \{[N', N, R'] \textbf{ in } Q\} \\ E41 & - := \{[N, [N', R']] \textbf{ in } E41\} \end{array}$$

Then the assignment to Q will be deleted by dead-code elimination, and the assignment to $E41$ is simply $E41\{N\} := \{\}$. This optimization can be applied to all algorithms derived above.

For complexity analysis, we only need to change formula (c4) to

$$\Sigma_{N \textbf{ in dom } O} \#E41\{N\} \qquad \text{(c4')}$$

Therefore,

$$\text{(c4')} \leq a * r \tag{38}$$

This does not change the overall asymptotic complexities.

For handling multiple queries, since this optimization updates $E41$ in the iteration, we need to preserve $E41$ after the initialization. To do this, we simply use a new set $E41'$ to function as $E41$ in the iteration: insert $E41' := E41$ immediately before the iteration, which can be a pointer assignment, and in the iteration, replace all uses of $E41$ by $E41'$. This does not change the complexity.

A specialization of constructor forms. Specializing the specification to our application enables some improvements.

The input is a regular tree grammar with productions P of the following forms built from a given program:

$$N' \to N, \quad N' \to c_i^{-1}(N), \quad N' \to [N]R'$$

and the following forms given by a query about the program:

$$N \to d, \quad N \to R$$

where R is as above, but R' is of forms l, $c(D, ..., D)$, $c(D, ..., D, N'', D, ..., D)$, and N''.

In the application, grammars are liveness patterns; based on their meaning as projection functions, a production of form $c(N_1, ..., N_k)$ can be replaced by at most k productions of forms $c(D, ..., D)$ and $c(D, ..., D, N'', D, ..., D)$, so we assume without loss of generality that R is of forms l, $c(D, ..., D)$, and $c(D, ..., D, N'', D, ..., D)$.

In the algorithm in Section 2, since $N \to d$ is not used to add other productions, and $N \to d$ is only added at the end if no $N \to R$ is in P, the third clause in the loop body can be replaced by

if P contains $N' \to c_i^{-1}(N)$ and $N \to c(D, ..., D, N'', D, ..., D)$, where N'' is in the ith position, add $N' \to N''$ to P

The specification, derivation, and analysis can be modified as follows. First, represent the right sides of special constructor forms as follows:

$c(D, ..., D)$	as	$[c]$
$c(D, ..., D, N'', D, ..., D)$, where N'' is in the ith position	as	$[c, i, N'']$

Then, the fixed-point expression is the same as in Section 2, except that the third clause in (4) becomes

$$\{[N', N''] : [N', C, I, N] \textbf{ in } Q,\ [N, [C, I, N'']] \textbf{ in } Q\}$$

The derivation of the algorithm is the same as above except that expensive subexpression $E3$ becomes

$$E3 = \{[N', N''] : [N', C, I, N] \textbf{ in } Q, [N, [C, I, N'']] \textbf{ in } Q\}$$

and its corresponding auxiliary expression becomes

$$E31 = \{[[C, I, N], N'] : [N', C, I, N] \textbf{ in } Q\}$$

In the loop body, the relevant updates are

if p is of form $[N, [C]]$, $\quad O$ **with** $:= [N, [C]]$;
$E1 \cup := \{[N', [C]] : N'$ **in** $E11\{N\}\}$;
same update to $E4$ as above

if p is of form $[N, [C, I, N'']]$, $\quad O$ **with** $:= [N, [C, I, N'']]$;
$E1 \cup := \{[N', [C, I, N'']] : N'$ **in** $E11\{N\}\}$;
$E3 \cup := \{[N', N''] : N'$ **in** $E31\{[C, I, N]\}\}$;
same update to $E4$ as above

and the corresponding result of finite differencing for case 3 becomes

$$
\begin{array}{l}
[N,[C,I,N'']]: \\
\quad E3 \cup:= \{[N',N''] : N' \textbf{ in } E31\{[C,I,N]\}\}; \\
\quad W \;\cup:= \{[N',N''] : N' \textbf{ in } E31\{[C,I,N]\} \mid [N',N''] \textbf{ notin } Q\};
\end{array}
$$

For initialization, the relevant updates are

$$
\begin{array}{ll}
\text{if } p \text{ is of form } [N',C,I,N], & \text{same updates to } E2 \text{ and } E21 \text{ as above} \\
& E3 \cup:= \{[N',N''] : [N,[C,I,N'']] \textbf{ in } Q\}; \\
& E31 \textbf{ with} := [[C,I,N],N'];
\end{array}
$$

and the corresponding code for case 5 becomes

$$
\begin{array}{l}
[N',C,I,N]: \\
\quad \text{same first three lines of this case as in (16)} \\
\quad E3 \cup:= \{[N',N''] : [N,[C,I,N'']] \textbf{ in } Q\}; \\
\quad W \cup:= \{[N',N''] : [N,[C,I,N'']] \textbf{ in } Q \mid [N'',N'] \textbf{ notin } Q\}; \\
\quad E31 \textbf{ with} := [[C,I,N],N'];
\end{array}
$$

Then, $E3$ and assignments to $E3$ are dead and eliminated, and uses of Q are replaced with uses of O and $E11$. We obtain the following case 5 in initialization:

$$
\begin{array}{l}
[N',C,I,N]: \\
\quad \text{same first two lines of this case as in (17)} \\
\quad W \cup:= \{[N',N''] : [N,[C,I,N'']] \textbf{ in } O \mid [N'',N'] \textbf{ notin } E11\}; \\
\quad E31 \textbf{ with} := [[C,I,N],N'];
\end{array}
$$

and the following case 3 in the loop body:

$$
\begin{array}{l}
[N,[C,I,N'']]: \\
\quad W \cup:= \{[N',N''] : N' \textbf{ in } E31\{[C,I,N]\} \mid [N'',N'] \textbf{ notin } E11\};
\end{array}
$$

The complexity analysis is the same as before except that (c3) is replaced with

$$\Sigma_{[N,[C,I,N'']] \textbf{ in } O} \#E31\{[C,I,N]\} \qquad \text{(c3')}$$

In the application, we have $\#E31\{[C,I,N]\} \leq 1$ for all $[C,I,N]$. So,

$$\text{(c3')} \leq 1 * \#O = \#O \tag{39}$$

Thus, without the above optimization to conditional forms, the sum of (c1) to (c5) is the same as before, and total complexities are also the same. With the optimization to conditional forms, the second way of estimating (c1) and (c5) gives $\mathcal{O}(\#O{*}g{+}r{*}g{*}a)$, and thus the sum of (c1) to (c5) is $\mathcal{O}(\min(\#O{*}h{+}r{*}a, \#O{*}g{+}r{*}g{*}a))$, and the total complexities are updated by replacing $\#O{*}\min(h{+}a, g{*}a)$ by $\min(\#O{*}h{+}r{*}a, \#O{*}g{+}r{*}g{*}a)$.

Complexity. The complexities are summarized in Table 1. If we could assume that a is a small constant

Algorithm	Complexity	With optimiz. to cond. forms & specializ. to cons. forms
(17), Sec. 4	$\mathcal{O}(\#O * \min(h+a, g*a) + \#P * g)$	$\mathcal{O}(\min(\#O * h + r * a, \#O * g + r * g * a) + \#P * g)$
(29), Sec. 5	$\mathcal{O}(\#O * \min(h+a, g*a) + \#P * g)$	$\mathcal{O}(\min(\#O * h + r * a, \#O * g + r * g * a) + \#P * g)$
(32), Sec. 5	$\mathcal{O}(\#O * \min(h+a, g*a) + \#P)$	$\mathcal{O}(\min(\#O * h + r * a, \#O * g + r * g * a) + \#P)$
(36), Sec. 5	$\mathcal{O}(\#O * \min(h+a, g*a))$ (simp.)	$\mathcal{O}(\min(\#O * h + r * a, \#O * g + r * g * a))$ (simp.)
	$\mathcal{O}(\#P_0)$ (init.)	$\mathcal{O}(\#P_0)$ (init.)

Table 1: Summary of Complexities.

(usually true, based on its meaning, experiments, and our experience), then with or without optimization to conditional forms and specialization of constructor forms, we obtain the second column in Table 2. If we could further assume that g is a constant (often true, suggested by its meaning, but more experiments are needed), then we obtain the last column. That is, the total complexity would be roughly linear in the size of the program, and simplification after a query would be roughly linear in the number of live program points.

Algorithm	Complexity if a is constant	If a and g are constants
(17), Sec. 4	$\mathcal{O}(\#O * \min(h,g) + \#P * g)$	$\mathcal{O}(r + \#P) = \mathcal{O}(\#P)$
(29), Sec. 5	$\mathcal{O}(\#O * \min(h,g) + \#P * g)$	$\mathcal{O}(r + \#P) = \mathcal{O}(\#P)$
(32), Sec. 5	$\mathcal{O}(\#O * \min(h,g) + \#P)$	$\mathcal{O}(r + \#P) = \mathcal{O}(\#P)$
(36), Sec. 5	$\mathcal{O}(\#O * \min(h,g))$ (simp.)	$\mathcal{O}(r)$ (simp.)

Table 2: Complexities If Certain Parameters Are Constants.

6 Lower-level implementation and experiments

We consider implementation of the two best algorithms, (32) for one query and (36) for multiple queries. The same data structures are suitable for both.

Low-level set operations. All the sets O, W, and $E11$ to $E41$ constructed in our algorithms are in fact maps, i.e., sets of pairs. To make this explicit, and to make each low-level operation simple, we do the following three groups of replacements in order:

1)	**while exists** Z **in** M
	    ...Z...
		with	**while exists** X **in dom** M
			    while exists Y **in** $M\{X\}$
			        ...$[X,Y]$...

2)	M **with** := $[X,Y]$	with	$M\{X\}$ **with** := Y
	M **less** := $[X,Y]$	with	$M\{X\}$ **less** := Y
	$[X,Y]$ **notin** M	with	Y **notin** $M\{X\}$
3)	S **with** := X	with	**if** X **notin** S S **with** := X

The first two groups clearly treat the domain of a map M as a set and the image of M at each element X as a set. The third guarantees that an addition is only for an element not located in the set; in general, similar replacements are done for deletions as well, but the only deletion in our algorithms is for an arbitrary element retrieved from the same set and thus already located in it. We do not need to transform **for**-loops in our algorithms, since they enumerate sets of tuples that are only read; we introduce pattern matching to make components of these tuples explicit, so other replacements apply in the loop body.

After the replacements, all the operations on all the domain and image sets are restricted to initializing to empty, retrieving an arbitrary element in a set by **for** or **while** or an indexed element by $T(I)$, adding an element not located in a set, deleting an element located in a set, and associative access, i.e., locating an element X in a set S (X **notin** S, or $M\{X\}$ where the domain of M is S). To support the complexity analysis in Sections 4 and 5, each of these operations needs to be done in $\mathcal{O}(1)$ time.

Data structure selection. Consider using a singly linked list for each of the domain and image sets of O, W, and $E11$ to $E41$. Let each element in a domain linked list contain a pointer to its image linked list, i.e., represent a map as a linked list of linked lists. It is easy to see that all operations except indexed retrieval and associative access can be done in worst-case $\mathcal{O}(1)$ time. The indexed retrievals are for tuples of arguments of constructor forms and are never updated and can be implemented using arrays. However, an associative access would take linear time if a linked list is naively traversed. A classical approach is to use hash tables [2] instead of linked lists. This gives average, rather than worst-case, $\mathcal{O}(1)$ time for each operation, and has an overhead of computing hashing related functions for each operation.

Paige et al. [40, 9] describe a technique for designing linked structures that support associative access in worst-case $\mathcal{O}(1)$ time with little space overhead for a general class of set-based programs. Consider associative accesses in the loop body below:

for X **in** W or **while exists** X **in** W
    ...X **in** S... or ...X **notin** S... or ...$M\{X\}$... where the domain of M is S

We want to locate value X in S after it has been located in W. The idea is to use a finite universal set B, called a base, to store values for both W and S, so that retrieval from W also locates the value in S. B

is represented as a set (this set is only conceptual) of records, with a K field storing the key (i.e., value). Set S is represented using a S field of B: records of B whose keys belong to S are connected by a linked list where the links are stored in the S field; records of B whose keys are not in S store a special value for undefined in the S field. Set W is represented as a separate linked list of pointers to records of B whose keys belong to W. Thus, an element of S is represented as *a field in* the record, and S is said to be *strongly based* on B; and element of W is represented as *a pointer to* the record, and W is said to be *weakly based* on B. This representation allows an arbitrary number of weakly based sets but only a constant number of strongly based sets. Essentially, base B provides a kind of indexing.

Our **while**-loop retrieves elements from the domain of W and locates these elements in the domains of O and $E11$ to $E41$. For example, at $O\{N\}$ in case 4 in the main loop, nonterminal N needs to be located in the domain of O. We use a base B for the set of nonterminals. The domain of W is weakly based on B, and the domains of O and $E11$ to $E41$ are strongly based on B. The only exception is that the domain of $E31$ needs a two-element key of the form $[C, N]$, but in the application, each N has only one corresponding C, so we simply use N as the key and record the corresponding C in a separate field to be checked against. It is worth noting that the linked lists for the domain (and images, discussed below) of W can be stacks, if retrieval (followed by deletion) and addition are both at the head of a list, or queues, if retrieval is at the head and addition is at the tail. The former corresponds to a depth-first search, and the latter corresponds to a breadth-first search; these are low-level decisions that only need to be made at the end.

Our algorithms test whether a value is not in the images of O, W, and $E11$ to $E41$ at any element in their domains, so there are $\mathcal{O}(n)$ sets that need to be strongly based, and thus the based-representation method does not apply here. We describe three representations for these images and discuss the trade-offs.

Data structure choices and trade-offs. The images of O, W, and $E11$ to $E41$ can be implemented using arrays, linked lists, hash tables, or a combination of linked lists and hash tables.

First, for the $\mathcal{O}(n)$ images of each of O, W, $E11$ to $E41$, we may make them strongly based using an array of fields. This includes making a base $B2$ for the set of good forms. Each membership test takes worst-case $\mathcal{O}(1)$ time and thus achieves the time complexities analyzed in Sections 4 and 5. However, this requires a total of quadratic space, which is not acceptable in practice for large programs. Quadratic initialization time can be avoided using the technique in [1, Exercise 2.12]; that technique can be implemented in a language like C but not Java or Scheme, since it requires allocating arrays without initializing them.

Second, we may use a singly linked list for each of the images of O, W, and $E11$ to $E41$. Such a list is called unbased representation [40] if it is a list of elements rather than a list of pointers to the elements in some base. Due to other associative accesses in the main loop body, any mention of a nonterminal (in images of W, $E11$, and $E21$, in domains of the images of $E31$, and in domains and images of the images of $E41$) should be implemented as a pointer to an element in base B. We also make a base $B2$ for the set of good forms (where nonterminals in the arguments of constructor forms are also implemented as pointers to elements in B), and represent any mention of a good form (in images of O and W and in images of the images of $E41$) as a pointer to an element in $B2$; use of $B2$ avoids an extra factor of a in the time complexity for comparing constructor forms if specialized constructor forms are not used. Linked-list representation incurs no asymptotic space overhead, but each membership test takes worst-case $\mathcal{O}(l)$ time where l is the length of such a linked list. Based on parameters introduced in Section 4, we know that $l = a$ for the images of $E21$, $E31$, and $E41$, $l = h$ for the images of $E11$, and $l = g$ for the images of O. Also, each element in W either has a right side in a good form or is a copy form, and thus $l = g + f$ for the images of W, where f is the dual of h, i.e., it is the maximum number of nonterminals to the right of a nonterminal:

$$f = \max{}_{N \textbf{ in dom } (\textbf{inv } E11)} \#(\textbf{inv } E11)\{N\} \tag{40}$$

In the application, f is bounded by the maximum of $g + 1$, the number of live call sites of any function, and the number of live occurrences of any formal parameter or bound variable. We count factor l in the complexity analysis and refine (c1) to (c8) for (i) as follows:

$$
\begin{array}{lll}
\text{cases 1-3:} & \Sigma_{[N,R] \textbf{ in } O,\ N' \textbf{ in } E11\{N\}} \#O\{N'\} + \#W\{N'\} & \text{(d1)} \\
& \Sigma_{[N,l] \textbf{ in } O,\ N' \textbf{ in } E21\{N\}} 1 + \#W\{N'\} & \text{(d2)} \\
& \Sigma_{[N,[C,T]] \textbf{ in } O,\ N' \textbf{ in } E31\{[C,N]\}} \#E11\{T(I)\} + \#W\{N'\} & \text{(d3)} \\
& \Sigma_{[N,R] \textbf{ in } O,\ [N',R'] \textbf{ in } E41\{N\}} (\textbf{if } R' \textit{ isR } \textbf{then } \#O\{N'\} \textbf{ else } \#E11\{R'\}) + \#W\{N'\} & \text{(d4)} \\
\text{case 4:} & \Sigma_{[N,N'] \textbf{ in } E11,\ R \textbf{ in } O\{N\}} \#O\{N'\} + \#W\{N'\} & \text{(d5)} \\
\text{case 5:} & \Sigma_{[N',C,I,N] \textbf{ in } P,\ l \textbf{ in } O\{N\}} 1 + \#W\{N'\} & \text{(d6)} \\
& \Sigma_{[N',C,I,N] \textbf{ in } P,\ [C,T] \textbf{ in } O\{N\}} \#E11\{T(I)\} + \#W\{N'\} & \text{(d7)} \\
\text{case 6:} & \Sigma_{[N',N,R'] \textbf{ in } P,\ R \textbf{ in } O\{N\}} (\textbf{if } R' \textit{ isR } \textbf{then } \#O\{N'\} \textbf{ else } \#E11\{R'\}) + \#W\{N'\} & \text{(d8)}
\end{array}
$$

Also, (ii) becomes $\Sigma_{p \textbf{ in } W} 1 + g + h$, rather than a only constant for each p **in** W, and (iii) becomes $\Sigma_{p \textbf{ in } P} 1 + a$, rather than a only constant for each p **in** P. We have, for (i),

$$
\begin{array}{l}
\text{(d1)} \le \text{(c1)} * (g + g + f) \\
\text{(d2)} \le \text{(c2)} * (1 + g + f) \\
\text{(d3)} \le \text{(c3)} * (h + g + f) \\
\text{(d4)} \le \text{(c4)} * (\max(g, h) + g + f) \\
\text{(d5)} \le \text{(c5)} * (g + g + f) \\
\text{(d6)} \le \text{(c6)} * (1 + g + f) \\
\text{(d7)} \le \text{(c7)} * (h + g + f) \\
\text{(d8)} \le \text{(c8)} * (\max(g, h) + g + f)
\end{array}
\tag{41}
$$

(ii) is still bounded by (i), and (iii) is $\mathcal{O}(\#P * a)$. For algorithms (32) and (36), the time for initialization is increased by a factor of a, and the time for the main loop is increased by a factor of $h + g + f$. This representation works well if h, g, and f are small. It works well for all our examples except a contrived worst-case example.

Third, we may maintain a hash table for each of the image sets. This achieves the time complexities analyzed in Sections 4 and 5, but they become average-case, rather than worst-case, complexities. It has relatively large overhead for small programs, and it faces the standard issues with hashing such as how to determine the size of hash tables and how to find good hash functions. Still, it works reasonably well for all our examples, though not as well as linked-list representation.

Finally, we can combine the use of linked lists with hash tables: use linked lists when the images are small, in particular for images of $E21$, $E31$, and $E41$, and possibly O, and use hash tables when the images are larger, in particular for images of $E11$ and W when they become large. This achieves the same complexities analyzed in Sections 4 and 5, also for average case.

Experiments. We implemented the simplification algorithm obtained from (35) with the optimization to conditional forms and used it to replace a previous algorithm in a prototype system for dead-code analysis and elimination [29]. The prototype system is implemented using the Synthesizer Generator [49], and the simplification algorithms are written in the Synthesizer Generator Scripting Language, STk, a dialect of Scheme. The labeled program, constructed grammar, and simplification result for the worst-case example in the Appendix are generated using the system.

We have used the system to analyze dozens of examples. Table 3 reports measurements of the most relevant parameters and simplification times from analyzing 14 programs with 25 different queries using the new simplification algorithm.

Programs `bigfun`, `minmax`, and `biggerfun` are the first few examples to illustrate dead-code analysis on recursive data in [29]. Program `worst`, given in the Appendix, is an example contrived to demonstrate the worst-case cubic-time complexity, and programs `worst10` and `worst20` are similar except that, instead of defining up to f_3, they define up to f_{10} and f_{20}, respectively. Programs `incsort` and `incout` are incremental programs for selection sort and outer product, respectively, derived using incrementalization [33], where dead code after incrementalization is to be eliminated. Programs `cachebin` and `cachelcs` are dynamic-programming programs transformed from straightforward exponential-time programs for binomial coefficients and longest common subsequences, respectively, using a method called cache-and-prune [31, 28], where cached intermediate results that are not used are to be pruned. Programs `calend`, `symbdiff`, `takr`, and `boyer` are taken from the Internet Scheme Repository [53]. Program `calend` is a collection of calendrical

program name	user query	#P	#O	n	r	a	h	g	f	c1,c5	c2	c3	c3'	c4	c4'	c	simp. time w/ GC	no GC
bigfun	lenf	48	47	36	23	2	2	3	3	40	0	4	1	24	14	68	.002	.001
minmax	getlen	112	89	81	31	3	2	5	11	76	0	8	2	48	23	132	.006	.005
minmax	getmin	112	149	81	49	3	2	8	11	129	2	38	14	72	33	241	.010	.007
biggerfun	evef	115	114	84	64	2	2	5	10	86	2	14	4	64	45	166	.008	.007
biggerfun	oddf	115	115	84	56	2	2	6	6	94	2	16	5	60	36	172	.008	.007
worst	f	28	69	24	24	2	4	4	4	64	0	0	0	21	12	85	.005	.004
worst10	f	70	419	59	59	2	11	11	11	407	0	0	0	133	33	540	.028	.018
worst20	f	130	1429	109	109	2	21	21	21	1407	0	0	0	463	63	1870	.097	.068
incsort	sort	144	132	108	49	3	2	11	5	139	2	20	9	98	29	259	.010	.007
incsort	sort'	144	33	108	24	3	2	5	5	24	6	0	0	15	11	45	.002	.001
incout	out	152	53	117	30	5	2	4	3	43	4	0	0	24	18	71	.003	.002
incout	out'	152	77	117	55	5	2	5	4	56	8	0	0	48	36	112	.005	.004
cachebin	bin	91	113	74	67	3	4	5	5	105	0	51	17	65	41	221	.009	.006
cachelcs	lcs	140	205	117	89	4	6	7	5	214	0	152	38	104	48	470	.018	.014
calend	gregorian-	1840	228	1551	192	5	12	4	25	178	0	66	30	115	111	359	.018	.015
calend	islamic-	1840	418	1551	346	5	12	4	25	339	4	144	68	199	189	686	.034	.024
calend	eastern-	1840	460	1551	375	5	24	4	25	380	4	186	89	207	197	777	.038	.030
calend	yahrzeit	1840	484	1551	428	5	11	4	25	373	0	108	46	293	290	774	.038	.030
calend	22 functions	1861	1604	1551	1352	5	37	4	25	1329	41	614	296	791	777	2775	.13	.10
symbdiff	deriv	1974	7636	1264	1221	3	65	13	65	11045	28	206	103	6639	855	17918	.59	.48
symbdiff	derivations-x	1974	7784	1264	1261	3	65	13	65	11214	30	206	103	6686	878	18136	.60	.48
takr	tak99	4005	2800	2804	2800	3	4	1	5	3000	0	0	0	2200	2200	5200	.23	.21
takr	run-takr	4005	2804	2804	2804	3	5	1	5	3004	0	0	0	2203	2203	5207	.23	.21
boyer	setup	4496	4513	4347	3755	3	106	8	6	1152	3496	1316	549	92	31	6056	.29	.23
boyer	setup,run-boyer	4497	39501	4347	4302	3	924	25	13	83925	3684	38370	18510	1377	254	127356	4.9	3.2

gregorian-: gregorian->absolute   islamic-: islamic-date   eastern-: eastern-orthodox-christmas

Table 3: Measurements for Example Programs.

functions [12]. Program `takr` is a 100-function version of TAK that tries to defeat cache memory effects. Program `symbdiff` does symbolic differentiation. Program `boyer` is a logic programming benchmark.

The queries are in the form $N \rightarrow l$, where N corresponds to the return value of a function in the second column of Table 3. In general, especially for libraries, such as the `calend` example, there may be multiple functions of interest; we included an example where we picked 22 functions at once.

The size of a program is captured by the total number of program points, n, which for pretty-printed programs is about twice the number of lines of code. We observe:

- $\#P$ ranges from $1.02n$ to $1.56n$.
- a is consistently very small.
- h varies widely.
- g and f are typically quite small.
- $\#O$ is roughly linear in r and in g.

Whether the observations about g and f hold for large programs need more experiments, but regardless, the measurements help confirm that the second way of estimating (c1) and (c5), not using h, better explains the running time in practice.

Parameters (c1) to (c5) are as defined in Section 4, (c4') and (c3') are as defined in Section 5, and $c =$ (c1) + (c2) + (c3) + (c4). Let (i1) to (i5) be the numbers of times the **if**-statement in (18) for the 1st, 3rd, 4th, 2nd, and 5th assignments to W, respectively, is executed. Our test for all the examples confirmed the following equalities:

$$\text{(i1)} + \text{(i5)} = \text{(c1)} = \text{(c5)}, \qquad \text{(i2)} = \text{(c2)}, \qquad \text{(i3)} = \text{(c3)}, \qquad \text{(i4)} = \text{(c4)}$$

and the relations in (22), (24), (38), and (39). We can see that (c1) and (c5) contribute most to c consistently except for the first query of `boyer`, and (c4) is more than (c2) and (c3) for all examples except `cachelcs` and `boyer`. We can see that c is roughly linear in $\#O$, with a small additional factor of g. The effects of optimization to conditional forms and specialization of constructor forms vary, but on average, (c4') is about 34% of (c4), and (c3') is about 40% of (c3).

The simplification time after initialization, in milliseconds, with and without garbage-collection time, is measured on a SUN station SPARC 20 with 60 MHz CPU and 256 MB main memory. The times in Table 3 are for when linked lists are used for images of O, W, and $E11$ to $E41$. We also measured the times for hash tables and for linked lists combined with hash tables, where images greater than 100 are switched to use hash tables; the former is about 50% slower than the number reported here, and the latter, only meaningful for `boyer`, is about 10% slower. The times reported are for with optimization to conditional forms but without specialization of constructor forms; the former gives up to 15% speedup, while the latter gives up to 10% speedup and often small slowdown.

We can see that the simplification time is very much linear in c, that is, it is roughly linear in $\#O$ with a small factor from g, and thus, it is linear in r and quadratic in g. Being close to linear in r rather than n is important, especially for analyzing libraries. Again, experiments measuring g for large programs are needed, but our measurements confirm the accurate complexities analyzed in terms of the identified parameters.

Liu and Stoller [29] reported measurements for n, r, $\#P$, $\#O$, and the simplification times for a subset of the examples. Our formal complexity analysis suggests that we should also measure a, h, g, f, and (c1) to (c5) accurately, which help confirm our complexity analysis and understand the running time in practice.

The analysis produces precise results, i.e., it finds all dead code, as desired. When used for eliminating dead code in deriving incremental programs [33], the speedup is often asymptotic. For example, dead code elimination enables incremental selection sort to improve from $\mathcal{O}(n^2)$ time to $\mathcal{O}(n)$ time. When adopted for pruning in the cache-and-prune method [31], the pruned programs consistently run faster, use less space, and are smaller in code size, often about 50%. It is also useful for analyzing benchmark programs. For example, for several functions in the `calend` program, only the slice for date, not year or month, is needed.

7 Discussion

Linear vs. cubic time complexity. Our higher-level transformations in Section 5 allow initialization to be done in $\mathcal{O}(\#P)$ time. Moreover, this only needs to be done while the productions are constructed. So we consider here only the main loop in algorithm (36).

Based on our experiments and experiences, it is realistic to consider a as a small constant and assume that each set associative operation can be implemented in constant time, at least on average. Thus, simplification after initialization takes $\mathcal{O}(\#O * \min(h, g))$ time. When h or g is a small constant, this is $\mathcal{O}(\#O)$, i.e., linear in terms of the output size, which would be optimal. Note, however, that O contains only productions in good forms, but productions of copy forms, resulted from selecting components of constructions and enabling conditions, are also added to the workset.

Our second way of estimating (c1) and (c5) yields the complexity $\mathcal{O}(\#O * g)$, which is $\mathcal{O}(r * g^2)$, not directly depending on h. Thus, when g is some small constant, the simplification time is $\mathcal{O}(r)$. The meaning of g suggests that g is often small, and our experiments with example programs support this, but more experiments are needed for large programs. Theoretically, in the worst case, simplification may take $\mathcal{O}(n^3)$ time, where r, h, and g are all of order n, and $\#O$ is of order n^2. We give an example below that exhibits this worst-case behavior. Identifying these parameters is a step forwards.

Linear vs. quadratic space complexity. In the worst case, the output O has quadratic size, so the worst-case space can be quadratic. However, using linked lists for the images of O, W, and $E11$ to $E41$ incurs virtually no space overhead, so the total space, besides $\mathcal{O}(\#P)$ for input related items, is $\mathcal{O}(\#O)$. Thus, when g is small, the space requirement is $\mathcal{O}(r)$.

Note that even though this representation increases the running time by a factor of $h+g+f$, our accurate analysis in (41) shows that (c1) and (c5) are increased only by a factor of $g+f$, not h; our experiments show that (c1) and (c5) dominate the cost c, and that g and f do not vary as widely as h. Again, a future work is to perform more measurements for large programs.

Using arrays to represent the images of O, W, and $E11$ to $E41$ always consumes the worst-case quadratic space. It has a large overhead, since for each array of size $O(n)$, basically only $O(g+h+f)$ entries are used. We think that using linked lists for small inputs and resorting to hash tables for larger inputs may be the best compromise.

A cubic-time quadratic-space worst-case program. To obtain a program whose simplification takes cubic time and quadratic space, we use three main ideas. First, an identity function id is defined and called a linear number of times, so that the liveness patterns at all calls to id are propagated to the arguments of all calls. Second, each call to id is the argument of a selector s, so that the liveness pattern N at the call to id is a distinct constructor form $c(N_1, ..., N_k)$ with a component N_i denoting the liveness pattern at the call to s. For simplicity, we use a constructor $pair$ with selectors fst and snd. Third, we define a linear number of functions f_1 to f_n, so that each f_i is a call to f_{i-1} applied to s applied to a call to id, and f_0 is just s applied to a call to id.

Finally, a function f simply calls f_n, and we query with the result of f being live. The Appendix gives such an example for which simplification of the generated constraints takes cubic time and quadratic space.

In such an example, factor g is linear, and the output O is quadratic. Note that simplification eliminates only extended forms; if further simplification could achieve minimization, then the output of this example may be made only linear. However, that would be a different analysis problem. In particular, minimization alone takes exponential time [15].

Language features affecting complexity. Again consider simplification after initialization and consider that a is a small constant. In the application, selector forms are constructed only from data construction in programs, and constructor forms are constructed only for selectors and tester; testers together generate no more than a constructor forms.

Suppose that the optimization to conditional forms is done, then in the second way of estimating (c1) and (c5), it is clear that (c1) and (c5) force the worst-case complexity $\mathcal{O}(\#O * g)$, where (c3) for selector forms is the sole contribution of factor $\#O$, while others contribute only factor r. Therefore, if there are no selector forms, i.e., in the application, there are no data constructions in the program, then the total complexity is $\mathcal{O}(r * g)$. Alternatively, if there are no constructor forms, i.e., in the application, there are no selections of components, then g is a constant, and the total complexity is $\mathcal{O}(r)$. This also means that the presence of conditional forms does not contribute to the super-linear complexity. The bottom line is that it is the precise analysis of recursive data structures, allowing both data constructions and selections, that causes the worst-case cubic-time complexity.

Pattern matching affecting clarity. Pattern matching was not discussed in the study of finite differencing, but it is not a language feature that introduces anything fundamentally new, so we do not need to invent new finite differencing techniques for it. However, it does help make the algorithm and derivation clear and simple. Without it, we would need to code a lot of tests and variable bindings explicitly; to reduce such code, we could group different productions of forms into different worksets and, in the **while**-loop, always pick p out of one of the groups until it is empty. This would make the algorithm and its derivation much less clear and much more clumsy.

8 Related work and conclusion

This work exploits the formal algorithm design and analysis method developed by Paige et al. that consists of dominated convergence [10], finite differencing [41, 39], and real-time simulation [40, 9]. Formal algorithm description and derivation allow us to obtain not only a more efficient algorithm but also more accurate complexity compared to previous work [29]. Previously, this method has been applied to a number of other nontrivial problems, e.g., [7, 26].

At the center of the method is incremental computation of expensive set expressions by finite differencing, where maintaining expensive subexpressions ($E1$ to $E4$ and W) and auxiliary expressions ($E11$ to $E41$), in additional to the desired output (O), is crucial for achieving the efficiency of the algorithm. The idea of exploiting and incrementally maintaining return value [33], intermediate results [32, 31], and auxiliary information [30] is described more explicitly by Liu et al. in the context of incrementalizing recursive functions [27]. Making these aspects explicit, as we did in this paper, help make the method for set expressions more systematic too.

Fixed-point expressions and dominated convergence [10] augment the method by specifying problems concisely at even higher levels and enabling higher level and more global transformations. Cai and Paige proposed transformations for simplifying initialization [10]; we propose more general transformations that make the overall code even smaller and, at the same time, allow more accurate complexity analysis. We also introduce pattern matching, which make the problem specification and algorithm derivation more succinct and easier to read.

Real-time simulation [40, 9] is a general method for implementing sets and set operations so that all associative accesses can be implemented in worst-case constant time. Unfortunately, real-time simulation using based representation, which incurs little space overhead, applies only partially to the constraint-simplification problem we consider. So we implement some sets as hash tables or linked lists whose lengths are typically small in our application. There might be deeper reasons and better solutions to this problem in general; the formal approach to data-structure selection in [18] might be helpful.

Our study of regular tree grammar based constraints for dead-code elimination is actually motivated by the need to prune unused values and computations in recursive functions and data structures after incrementalization [32, 31]. Previously, an algorithm was designed by informally applying finite differencing so that after adding a new production, we consider only productions in extended forms whose right sides use the left-side symbol of the new production [29]; a simple analysis gives a $\mathcal{O}(n^3)$ time complexity, while experiments showed that the running time for the simplification was roughly $\mathcal{O}(r)$. Implementation of the algorithm in this paper was found to be two to ten times as fast as the previous algorithm.

Regular tree grammar based constraints have been used for analyzing recursive data in other applications and go back at least to Reynolds [51] and Schwartz [54]. Related work includes flow analysis for memory optimization by Jones and Muchnick [25], binding-time analysis for partial evaluation by Mogensen [36], set-based analysis of ML by Heintze [20], type inference by Aiken et al. [3, 4], backward slicing by Reps and Turnidge [50], and set-based analysis for debugging Scheme by Flanagan and Felleisen [16]. Some of these are general type inference and are only shown to be decidable [4] or take exponential time in the worst case [3]. For others, either a cubic time complexity is given based on a simple worst-case analysis of a relatively straightforward algorithm [20, 16], or algorithm complexity is not discussed explicitly [25, 36, 50].

Constraints have also been used for other analyses, in particular, analyses handling mainly higher-order functions or pointers. This includes higher-order binding-time analysis by Henglein [24], Bondorf and Jørgensen [8], and Birkedal and Welinder [5, 6], points-to analysis by Steensgaard [58], and control flow analysis for special cases by Heintze and McAllester [22]. The last restricts type sizes and has a linear time complexity, and the others use union-find algorithms [24] and have an almost linear time complexity. These complexities are practical for analyzing large programs, but these analyses either do not consider recursive data structures [24, 58], or use bounded domains [8, 5, 6, 22] and are thus less precise than grammar constraints constructed based on uses of recursive data in their contexts.

People study methods to speed up the cubic-time analysis algorithms. For example, Heintze [19] describes implementation techniques such as dependency directed updating and special representations, which has the same idea as incremental update by finite differencing and efficient access by real-time simulation. Flanagan and Felleisen [16] study techniques for component-wise simplification. Fähndrich et al. [13] study a technique for eliminating cycles in the inclusion constraint graphs. Su et al. [59] study techniques for reducing redundancies caused by transitivity in the constraint graphs. These improvements are all found to be very effective. Moreover, sometimes a careful implementation of a worst-case cubic-time [21, 29] (or quadratic-time [57]) analysis algorithm seems to give nearly linear behavior [21, 57, 29]. Our work in this paper is a start in the formal study of the reasons.

In particular, our analysis also adds edges dynamically, through selecting components of constructions and enabling conditions, and our application also has the cycle and redundancy problems caused by dynamic transitivity, as studied in [13, 59]. However, our algorithm still proceeds in a linear fashion. That is, if we have constraints $N_1 \to N_2, ..., N_{k-1} \to N_k$, possibly some added dynamically, and possibly $N_i = N_j$ for some i, j such that $1 \leq i \leq j \leq k$, we do not add any edges $N_i \to N_j$ for any i, j such that $1 \leq i \leq j \leq k$; only when a new $N_k \to R$ is added, we add an $N_{k-1} \to R$ if it is not already added and subsequently an $N_{k-2} \to R$ and so on. This formalizes Heintze's algorithm [19]. For comparison, a future work would be to formalize the algorithms in [13, 59]. As our problem is equivalent to computing Datalog queries, McAllester's complexity results for Datalog queries [34] are related as well. It will also be interesting to formalize and compare with [60].

The possibly non-linear behavior of this problem is because each nonterminal N_i may go to a linear number of good forms. This is so even if we collapse strongly connected components of nonterminals that go to one another and thus go to the same good forms; our worst-case example in the Appendix supports this. Heintze and McAllester [23] show that it is unlikely we can improve the cubic-time worst-case complexity for certain flow analysis and subtyping problems, because it is as hard as the 2-NPDA acceptability problem, for which no faster algorithm has been found for over 30 years. Melski and Reps [35] also give some insight on the cubic-time complexity by showing the interconvertibility of a class of set constraints and context-free language reachability [48, 46, 47].

To summarize, for the problem of dead-code elimination on recursive data, this paper shows that formal specification, design, and analysis lead to an efficient algorithm with exact complexity factors that can be measured in experiments. Clearly, there is a large body of work on all kinds of program analysis algorithms [37], from type inference algorithms, e.g., [45], to efficient fixed-point computation, e.g., [14]. Precise and unified specification, design, and complexity analysis of all kinds of program analysis algorithms deserve much further study. We believe that such study can benefit greatly from the approach of Paige et al. [41, 39, 10, 40, 9], as illustrated in this work, and from the more formal characterization by Goyal [18].

Appendix: A worst-case program

Program. A program is a set of recursive function definitions, together with a set of constructor definitions, each with the corresponding tester and selectors.

$$
\begin{aligned}
f(x) &= f_3(x);\\
id(x) &= x;\\
f_0(x) &= fst(id(x));\\
f_1(x) &= f_0(fst(id(x)));\\
f_2(x) &= f_1(fst(id(x)));\\
f_3(x) &= f_2(fst(id(x)));\\
\\
pair &: pair?(fst, snd);
\end{aligned}
$$

Labeled program. The program is labeled, with a distinct nonterminal associated with each program point, as follows:

$$
\begin{aligned}
f(^{N_{24}:}x) &=^{N_{23}:} f_3(^{N_{22}:}x);\\
id(^{N_{21}:}x) &=^{N_{20}:} x;\\
f_0(^{N_{19}:}x) &=^{N_{18}:} fst(^{N_{17}:}id(^{N_{16}:}x));\\
f_1(^{N_{15}:}x) &=^{N_{14}:} f_0(^{N_{13}:}fst(^{N_{12}:}id(^{N_{11}:}x)));\\
f_2(^{N_{10}:}x) &=^{N_{9}:} f_1(^{N_{8}:}fst(^{N_{7}:}id(^{N_{6}:}x)));\\
f_3(^{N_{5}:}x) &=^{N_{4}:} f_2(^{N_{3}:}fst(^{N_{2}:}id(^{N_{1}:}x)));
\end{aligned}
$$

Constructed grammar. The grammar constructed from the given program is

$$
\begin{aligned}
&N_{24}\to N_{22}, N_{22}\to [N_{23}]N_5, N_4\to N_{23}, N_{21}\to N_{20},\\
&N_{19}\to N_{16}, N_{16}\to [N_{17}]N_{21}, N_{20}\to N_{17}, N_{17}\to [N_{18}]pair(N_{18}, N_0),\\
&N_{15}\to N_{11}, N_{11}\to [N_{12}]N_{21}, N_{20}\to N_{12}, N_{12}\to [N_{13}]pair(N_{13}, N_0), N_{13}\to [N_{14}]N_{19}, N_{18}\to N_{14},\\
&N_{10}\to N_6, N_6\to [N_7]N_{21}, N_{20}\to N_7, N_7\to [N_8]pair(N_8, N_0), N_8\to [N_9]N_{15}, N_{14}\to N_9,\\
&N_5\to N_1, N_1\to [N_2]N_{21}, N_{20}\to N_2, N_2\to [N_3]pair(N_3, N_0), N_3\to [N_4]N_{10}, N_9\to N_4, N_0\to D
\end{aligned}
$$

User query. A user query is

$$N_{23}\to L$$

Simplification result. Clearly, N_2, N_7, N_{12}, N_{17} at all calls to id, i.e., at arguments of calls to fst, have liveness patterns $pair(N_3, N_0)$, $pair(N_8, N_0)$, $pair(N_{13}, N_0)$, $pair(N_{18}, N_0)$, respectively. Then N_1, N_6, N_{11}, N_{16} at arguments of all calls to id each goes to all these right sides. These right sides are also included in the liveness patterns at formal parameters of all f_i's, since in each f_i they are for the same variable x, and finally, these right sides are also included in the liveness patterns at arguments of all calls to f_i's. The output of simplification, sorted by nonterminal numbers, is

$$
\begin{aligned}
&N_{24}\to pair(N_{18}, N_0), N_{24}\to pair(N_{13}, N_0), N_{24}\to pair(N_8, N_0), N_{24}\to pair(N_3, N_0),\ \ N_{23}\to L,\\
&N_{22}\to pair(N_{18}, N_0), N_{22}\to pair(N_{13}, N_0), N_{22}\to pair(N_8, N_0), N_{22}\to pair(N_3, N_0),\\
&N_{21}\to pair(N_{18}, N_0), N_{21}\to pair(N_{13}, N_0), N_{21}\to pair(N_8, N_0), N_{21}\to pair(N_3, N_0),\\
&N_{20}\to pair(N_{18}, N_0), N_{20}\to pair(N_{13}, N_0), N_{20}\to pair(N_8, N_0), N_{20}\to pair(N_3, N_0),\\
&N_{19}\to pair(N_{18}, N_0), N_{19}\to pair(N_{13}, N_0), N_{19}\to pair(N_8, N_0), N_{19}\to pair(N_3, N_0),\ \ N_{18}\to L,\\
&N_{17}\to pair(N_{18}, N_0),\\
&N_{16}\to pair(N_{18}, N_0), N_{16}\to pair(N_{13}, N_0), N_{16}\to pair(N_8, N_0), N_{16}\to pair(N_3, N_0),\\
&N_{15}\to pair(N_{18}, N_0), N_{15}\to pair(N_{13}, N_0), N_{15}\to pair(N_8, N_0), N_{15}\to pair(N_3, N_0),\ \ N_{14}\to L,\\
&N_{13}\to pair(N_{18}, N_0), N_{13}\to pair(N_{13}, N_0), N_{13}\to pair(N_8, N_0), N_{13}\to pair(N_3, N_0),\\
&N_{12}\to pair(N_{13}, N_0),\\
&N_{11}\to pair(N_{18}, N_0), N_{11}\to pair(N_{13}, N_0), N_{11}\to pair(N_8, N_0), N_{11}\to pair(N_3, N_0),\\
&N_{10}\to pair(N_{18}, N_0), N_{10}\to pair(N_{13}, N_0), N_{10}\to pair(N_8, N_0), N_{10}\to pair(N_3, N_0),\ \ N_9\to L,\\
&N_8\to pair(N_{18}, N_0), N_8\to pair(N_{13}, N_0), N_8\to pair(N_8, N_0), N_8\to pair(N_3, N_0),\\
&N_7\to pair(N_8, N_0),\\
&N_6\to pair(N_{18}, N_0), N_6\to pair(N_{13}, N_0), N_6\to pair(N_8, N_0), N_6\to pair(N_3, N_0),\\
&N_5\to pair(N_{18}, N_0), N_5\to pair(N_{13}, N_0), N_5\to pair(N_8, N_0), N_5\to pair(N_3, N_0),\ \ N_4\to L,\\
&N_3\to pair(N_{18}, N_0), N_3\to pair(N_{13}, N_0), N_3\to pair(N_8, N_0), N_3\to pair(N_3, N_0),\\
&N_2\to pair(N_3, N_0),\\
&N_1\to pair(N_{18}, N_0), N_1\to pair(N_{13}, N_0), N_1\to pair(N_8, N_0), N_1\to pair(N_3, N_0),\ \ N_0\to D
\end{aligned}
$$

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass., 1983.

[3] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1991.

[4] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1994.

[5] L. Birkedal and M. Welinder. Binding-time analysis for standard ML. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical Report 94/9*, pages 61–71. Department of Computer Science, The University of Melbourne, June 1994.

[6] L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, Sept. 1995.

[7] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3):189–220, 1995.

[8] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[9] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*, pages 126–164. North-Holland, Amsterdam, 1991.

[10] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.

[11] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 170–181. ACM, New York, June 1995.

[12] N. Dershowitz and E. M. Reingold. Calendrical calculations. *Software—Practice and Experience*, 20(9):899–928, Sept. 1990.

[13] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96. ACM, New York, June 1998.

[14] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In C. Hankin, editor, *Proceedings of the 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, Berlin, 1998.

[15] C. Flanagan. *Effective Static Debugging via Componential Set-Based Program Analysis*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, May 1997.

[16] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, Mar. 1999.

[17] F. Gecseg and M. Steinb. *Tree Automata*. Akademiai Kiado, Budapest, 1984.

[18] D. Goyal. *A Language Theoretic Approach to Algorithms*. PhD thesis, Department of Computer Science, New York University, Jan. 2000.

[19] N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779. The MIT Press, Cambridge, Mass., Nov. 1992.

[20] N. Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317. ACM, New York, June 1994.

[21] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer-Verlag, Berlin, 1994.

[22] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. ACM, New York, June 1997.

[23] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 342–351. IEEE CS Press, Los Alamitos, Calif., 29 June-2 July 1997.

[24] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer-Verlag, Berlin, Aug. 1991.

[25] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, pages 102–131. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[26] J. P. Keller and R. Paige. Program derivation with verified transformations - a case study. *Communications on Pure and Applied Mathematics*, 48(9-10):1053–1113, 1995.

[27] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, 2000.

[28] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 288–305. Springer-Verlag, Berlin, Mar. 1999.

[29] Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In SAS 1999 [52], pages 211–231.

[30] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In POPL 1996 [44], pages 157–170.

[31] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

[32] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In PEPM 1995 [43], pages 190–201.

[33] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[34] D. McAllester. On the complexity analysis of static analyses. In SAS 1999 [52], pages 312–329.

[35] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoret. Comput. Sci.*, 248(1-2), Nov. 2000.

[36] T. Mogensen. Separating binding times in language specifications. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 12–25. ACM, New York, Sept. 1989.

[37] F. Nielson, H. R. Nielson, and C. Hankin, editors. *Principles of Program Analysis.* Springer-Verlag, 1981.

[38] R. Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 6 of *Computer Science and Artificial Intelligence.* UMI Research Press, Ann Arbor, Michigan, 1981. Revision of Ph.D. dissertation, New York University, 1979.

[39] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.

[40] R. Paige. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*, pages 69–73. Canadian Scholars Press, 1989. Proceedings of ICCI '89: The International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

[41] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[42] R. Paige and Z. Yang. High level reading and data structure compilation. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 456–469. ACM, New York, Jan. 1997.

[43] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM, New York, June 1995.

[44] *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, Jan. 1996.

[45] J. Rehof. *The Complexity of Simple Subtyping Systems.* PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, Apr. 1998.

[46] T. Reps. Shape analysis as a generalized path problem. In PEPM 1995 [43], pages 1–11.

[47] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, Nov. 1998. Special issue on program slicing.

[48] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, Jan. 1995.

[49] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1988.

[50] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, Berlin, 1996.

[51] J. C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68: Proceedings of IFIP Congress 1968*, volume 1, pages 456–461. North-Holland, Amsterdam, 1969.

[52] *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Sept. 1999.

[53] The Internet Scheme Repository. http://www.cs.indiana.edu/scheme-repository/.

[54] J. T. Schwartz. Optimization of very high level languages – I: Value transmission and its corollaries. *Journal of Computer Languages*, 1(2):161–194, 1975.

[55] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, Berlin, New York, 1986.

[56] W. K. Snyder. The SETL2 Programming Language. Technical report 490, Courant Institute of Mathematical Sciences, New York University, Sept. 1990.

[57] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In T. Gyimothy, editor, *Proceedings of the 6th International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, Berlin, 1996.

[58] B. Steensgaard. Points-to analysis in almost linear time. In POPL 1996 [44], pages 32–41.

[59] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Conference Record of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 81–95. ACM, New York, Jan. 2000.

[60] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.