Solving Regular Tree Grammar Based Constraints*

Yanhong A. Liu[1], Ning Li[2], and Scott D. Stoller[1]

[1] Computer Science Dept., State University of New York, Stony Brook, NY 11794
[2] Computer Science Dept., University of Wisconsin, Madison, WI 53706
{liu,stoller}@cs.sunysb.edu, ning@cs.wisc.edu

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 propose 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 [22, 29, 17, 37, 25]. 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 [31, 34, 32], 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 [8] to transform fixed-point expressions into loops, (2) finite differencing [34, 32] to transform expensive set expressions in loops into incremental

* 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.

operations, and (3) real-time simulation at the lower level [33, 7] 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 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 [33] 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 [25].

The main contributions of this work include

(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 all kinds of program manipulation, and static analysis is naturally described as computing fixed points, 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, and 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 1.

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 [25].[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.

[1] The presentation here includes minor notational changes and simplifications. In particular, in [25], the condition in the first production for a binding expression is unnecessary.

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 may 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.

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

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 $N \rightarrow l$ 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. Appendix A gives a small example program together with the constructed grammar, a user query, and the simplification result.

All the production forms here are the same as or similar to those studied by many people. For example, standard forms are as in [14, 22, 9], copy forms are common in grammars, selector forms are first seen in [22], and conditional forms have counterparts in [3, 17]. Overall, the constraints here extend those by Jones and Muchnick [22].

Notation. We use a set-based language. It is based on SETL [41, 42] extended with a fixed-point operation by Cai and Paige [8]; 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 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. 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** $S \mid Z\}$

$\mathbf{LFP}_{\subseteq,X}(F(Y), Y)$ denotes the minimum element Y, with respect to partial ordering $\subseteq$, that satisfies the condition $X \subseteq Y$ and $F(Y) = Y$. 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. We represent the right-side R' forms as follows:

$$\begin{array}{ll} l & \text{as } l, \text{ where } l \text{ is a special constant} \\ c(N_1, ..., N_k) & \text{as } [c, [N_1, ..., N_k]] \\ N & \text{as } N \end{array} \tag{2}$$

and represent the productions as follows:

$$\begin{array}{ll} N' \rightarrow R' & \text{as } [N', \text{representation of } R'] \\ N' \rightarrow c_i^{-1}(N) & \text{as } [N', c, i, N] \\ N' \rightarrow [N]R' & \text{as } [N', N, \text{representation of } R'] \end{array} \tag{3}$$

This representation allows us to distinguish all the production forms by simple pattern matching against constants and tuples of different lengths. We also need to tell whether an R' form is an R form or an N form, so for convenience, we define:

$$\begin{array}{l} R' \ isR = R' \textbf{ of } l \textbf{ or } R' \textbf{ of } [C, T] \\ R' \ isN = \textbf{not } (R' \textbf{ of } l \textbf{ or } R' \textbf{ of } [C, T]) \end{array} \tag{4}$$

The simplification algorithm in (1) can be specified as follows. The input is a set P of productions in the new representation. The **repeat**-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:

$$\begin{array}{rl} F(Q) = & \{[N', R] : [N', N] \textbf{ in } Q,\ [N, R] \textbf{ in } Q \mid R \ isR\} \cup \\ & \{[N', l] : [N', C, I, N] \textbf{ in } Q,\ [N, l] \textbf{ in } Q\} \cup \\ & \{[N', T(I)] : [N', C, I, N] \textbf{ in } Q, [N, [C, T]] \textbf{ in } Q\} \cup \\ & \{[N', R'] : [N', N, R'] \textbf{ in } Q,\ [N, R] \textbf{ in } Q \mid R \ isR\} \end{array} \tag{5}$$

Since $F(Q) \subseteq Q$ iff $F(Q) \cup Q = Q$, the loop computes

$$\mathbf{LFP}_{\subseteq,P}(F(Q) \cup Q,\ Q) \tag{6}$$

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

$$O = \{[N, R] \textbf{ in } \mathbf{LFP}_{\subseteq,P}(F(Q) \cup Q,\ Q) \mid R \ isR\} \tag{7}$$

Note that $G = \lambda Q.F(Q) \cup Q$ is monotone, i.e., if $Q_1 \subseteq Q_2$ then $G(Q_1) \subseteq G(Q_2)$, and is inflationary at P, i.e., $P \subseteq G(P)$.

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 systematically manipulating them, uniform notation helps.

3 Approach

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

Dominated convergence [8] 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 $\mathbf{LFP}_{\subseteq,P}(F(Q) \cup Q,\ Q)$ in (7) is transformed into the following **while**-loop, making use of $\lambda Q.F(Q) \cup Q$ being monotone and inflationary at P:

$$\begin{array}{l} Q := P; \\ \textbf{while exists } p \textbf{ in } F(Q) - Q \\ \quad Q \textbf{ with} := p; \end{array} \tag{8}$$

This code is followed by

$$O = \{[N, R] \textbf{ in } Q \mid R\ isR\}; \tag{9}$$

Finite differencing [34, 32] transforms expensive set operations in a loop into incremental operations. The idea is to replace expensive expressions exp_1, ..., exp_n in a loop $LOOP$ with fresh variables E_1, ..., E_n, respectively, and maintain the invariants $E_1 = exp_1$, ..., $E_n = exp_n$ by inserting appropriate initializations or updates to E_1, ..., E_n at each assignment in $LOOP$. We denote the transformed loop as

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

For our program (8) and (9) from Step 1, expensive expressions, i.e., non-constant-time expressions here, are the one that computes O and others that are needed for computing $F(Q) - Q$. We use fresh variables to hold their values. These variables are initialized together with the assignment $Q := P$ and are updated incrementally as Q is augmented by p in each iteration. Liu [23] gives references to much work that exploited related ideas.

Real-time simulation [33, 7] 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.

4 Finite differencing

Identifying expensive subexpressions. The output O in (9) 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 (5), and W, the workset:

$$\begin{array}{l} E1 = \{[N', R] : [N', N] \textbf{ in } Q, [N, R] \textbf{ in } Q \mid R\ isR\} \\ E2 = \{[N', l] : [N', C, I, N] \textbf{ in } Q, [N, l] \textbf{ in } Q\} \\ E3 = \{[N', T(I)] : [N', C, I, N] \textbf{ in } Q, [N, [C, T]] \textbf{ in } Q\} \\ E4 = \{[N', R'] : [N', N, R'] \textbf{ in } Q, [N, R] \textbf{ in } Q \mid R\ isR\} \\ W\ = F(Q) - Q = E1 \cup E2 \cup E3 \cup E4 - Q \end{array} \tag{10}$$

Thus, the overall computation becomes

$$\begin{array}{l} \Delta O, E1, E2, E3, E4, W \ \langle \ Q := P; \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{while exists } p \textbf{ in } W \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad Q \textbf{ with} := p; \ \rangle \end{array} \tag{11}$$

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 $N' \rightarrow [N]R'$.

$$\begin{array}{l} E11 = \{[N, N'] : [N', N] \textbf{ in } Q \mid N \ isN\} \\ E21 = \{[N, N'] : [N', C, I, N] \textbf{ in } Q\} \\ E31 = \{[[C, N], [N', I]] : [N', C, I, N] \textbf{ in } Q\} \\ E41 = \{[N, [N', R']] : [N', N, R'] \textbf{ in } Q\} \end{array} \tag{12}$$

These expressions are introduced for differentiating $E1$ to $E4$, respectively. For example, $E11$ is introduced for differentiating $E1$ in (10) 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 (10): 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

$$\begin{array}{l} \Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \ \langle \ Q := P; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{while exists } p \textbf{ in } W \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Q \textbf{ with} := p; \ \rangle \end{array} \tag{13}$$

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 (13):

$$\Delta O, E1, E2, E3, E4, W, E11, E21, E31, E41 \ \langle \ Q \textbf{ with} := p; \ \rangle \tag{14}$$

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 determine how the sets O, $E1$ to $E4$, and $E11$ to $E41$ are updated. 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 obtain

the following complete code for the loop body:

```
Q with := p;
W less := p;
case p of
   [N,R], where R isR :                                  //if p is N→R
      O with := [N,R];
      E1 ∪:= {[N',R] : N' in E11{N}};                    //add N'→R for all N'→N
      W  ∪:= {[N',R] : N' in E11{N} | [N',R] notin Q};
      E4 ∪:= {[N',R'] in E41{N}};                        //add N'→R' for all N'→[N]R'
      W  ∪:= {[N',R'] in E41{N} | [N',R'] notin Q};
   [N,l] :                                               //if p is N→l
      E2 ∪:= {[N',l] : N' in E21{N}};                    //add N'→l for all N'→C_I^{-1}(N)
      W  ∪:= {[N',l] : N' in E21{N} | [N',l] notin Q};
   [N,[C,T]] :                                           //if p is N→C(T(1),...,T(k))
      E3 ∪:= {[N',T(I)] : [N',I] in E31{[C,N]}};         //add N'→T(I) for all N'→C_I^{-1}(N)
      W  ∪:= {[N',T(I)] : [N',I] in E31{[C,N]} | [N',T(I)] notin Q};
   [N',N], where N isN :                                 //if p is N'→N
      E1 ∪:= {[N',R] : R in O{N}};                       //add N'→R for all N→R
      W  ∪:= {[N',R] : R in O{N} | [N',R] notin Q};
      E11 with := [N,N'];
```

(15)

These updates are keys for achieving high efficiency: after adding a new production, we consider only productions that are directly affected.

Initialization. Sets O, $E1$ to $E4$, W, and $E11$ to $E41$ need to be initialized together with $Q := P$ in (13). To do this, we add each p from P into Q one by one, and update each of these sets incrementally as in the loop body. We have the same four cases of p as in the loop body (15) and the cases for two additional forms of p, namely $[N', C, I, N]$ and $[N', N, R]$. We obtain the following complete code for initialization:

```
O,E1,E2,E3,E4,W,E11,E21,E31,E41,Q := {};
for p in P
   Q with := p;
   W less := p;
   case p of
      same four cases of p as in the loop body
      [N',C,I,N] :                                       //if p is N'→C_I^{-1}(N)
         E2 ∪:= {[N',l] : l in Q{N}};                    //add N'→l for all N→l
         W  ∪:= {[N',l] : l in Q{N} | [N',l] notin Q};
         E21 with := [N,N'];
         E3 ∪:= {[N',T(I)] : [C,T] in Q{N}};             //add N'→T(I) for all N→C(T(1),...,T(k))
         W  ∪:= {[N',T(I)] : [C,T] in Q{N} | [N',T(I)] notin Q};
         E31 with := [[C,N],[N',I]];
      [N',N,R'] :                                        //if p is N'→[N]R'
         E4 ∪:= {[N',R'] : R in Q{N} | R isR};           //add N'→R' for all N→R
         W  ∪:= {[N',R'] : R in Q{N} | R isR,[N',R'] notin Q};
         E41 with := [N,[N',R']];
```

(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:

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

We obtain the following complete algorithm:

```
O, W, E11, E21, E31, E41 := {};
for p in P
  W less := p;
  case p of
    same four cases of p as in the loop body
    [N', C, I, N] :
      W ∪:= {[N', l] : l in O{N} | [N', l] notin O};
      E21 with := [N, N'];
      W ∪:= {[N', T(I)] : [C, T] in O{N} | [T(I), N'] notin E11};
      E31 with := [[C, N], [N', I]];
    [N', N, R'] :
      W ∪:= {[N', R'] : R in O{N} | if R' isR then [N', R'] notin O else [R', N'] notin E11};
      E41 with := [N, [N', R']];
while exists p in W                                                          (17)
  W less := p;
  case p of
    [N, R], where R isR :
      O with := [N, R];
      W ∪:= {[N', R] : N' in E11{N} | [N', R] notin O};
      W ∪:= {[N', R'] in E41{N} | if R' isR then [N', R'] notin O else [R', N'] notin E11};
    [N, l] :
      W ∪:= {[N', l] : N' in E21{N} | [N', l] notin O};
    [N, [C, T]] :
      W ∪:= {[N', T(I)] : [N', I] in E31{[C, N]} | [T(I), N'] notin E11};
    [N', N], where N isN :
      W ∪:= {[N', R] : R in O{N} | [N', R] notin O};
      E11 with := [N, N'];
```

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

```
for Y in S                                                                   (18)
  if Z then
    W with := X;
```

Complexity analysis. 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; Section 6 describes how to achieve this. Other operations 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.
  Meaning: In the application, 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.

– 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 $\#O \leq r * g$.

– 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 $\#P \leq n * a$.
Parameter n is not used in the precise 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). The other cases are similar.

Using the parameters introduced above, we have

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

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

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

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,

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

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; if we assume there is a constant number of them, then (c6) to (c8) are $\mathcal{O}(\#P)$, and the total complexity is $\mathcal{O}(\#O * \min(h + a, g * a) + \#P)$.

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 [8] proposed a high-level transformation that can drastically simplify the initialization and do all the work in the loop body. By Theorem 5 in [8], the fixed-point expression (6) is equivalent to

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

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. However, this merging reduces the accuracy of the complexity analysis. The complexity analysis is similar to that in Section 4. 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, because (c6) to (c8) are now from the main loop, where g is not bounded by a constant.

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. Our method is supported by the following theorem.

Theorem 1. *For all* $P_0 \subseteq P$, $\mathbf{LFP}_{\subseteq,P_0}((P - P_0) \cup F(Q) \cup Q, Q)$ *exists if and only if* $\mathbf{LFP}_{\subseteq,P}(F(Q) \cup Q, Q)$ *exists, and if they exist, they are equal.*

Proof. $\mathbf{LFP}_{\subseteq,P_0}((P - P_0) \cup F(Q) \cup Q, Q) = \mathbf{LFP}_{\subseteq,\{\}}(P_0 \cup (P - P_0) \cup F(Q) \cup Q, Q)$
$= \mathbf{LFP}_{\subseteq,\{\}}(P \cup F(Q) \cup Q, Q) = \mathbf{LFP}_{\subseteq,P}(F(Q) \cup Q, Q)$. □

We apply Theorem 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 (6) is equivalent to $\mathbf{LFP}_{\subseteq,P_0}(P - P_0 \cup F(Q) \cup Q,\ Q)$. Transforming this into a **while**-loop and applying finite differencing yields 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 $P - P_0$:

```
O, W, E11, E21, E31, E41 := {};
for p in P
    case p of
        [N', C, I, N] :
            E21 with := [N, N'];
            E31 with := [[C, N], [N', I]];
        [N', N, R'] :
            E41 with := [N, [N', R']];
        other :
            W with := p;
same iteration as in algorithm (17)
```

(29)

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

$$\mathcal{O}(\#O * \min(h + a, g * a) + \#P) \tag{30}$$

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

Handling multiple queries. In the application, 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 1, initialization using P_0 followed by simplification using P_1 can be specified as

$$\mathbf{LFP}_{\subseteq,P_0}(P_1 \cup F(Q) \cup Q,\ Q) \tag{31}$$

which is transformed into

```
Q := P0;
while exists p in P1 ∪ F(Q) − Q
    Q with := p;
```

(32)

Applying finite differencing in a similar way as above yields an algorithm that takes

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

time for simplification after a query.

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:

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

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, (c4') $\leq a*r$. 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.

6 Lower-level implementation and experiments

We consider implementation of the two best algorithms, (29) for one query and the algorithm obtained from (32) for multiple queries. The same data structures for representing sets are suitable for both. All sets involved are clearly finite based on the analysis in Sections 4 and 5.

Low-level set operations. All the sets constructed in our algorithms are in fact maps, i.e., sets of pairs. To make this explicit, we do the following three groups of replacements in order:

1) **while exists** Z **in** M	with	**while exists** X **in dom** M
$...Z...$		**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 set operations are restricted to those described in Section 4, with the above guarantees about elements added or deleted. 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 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 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. [33, 7] describe a technique for designing linked structures that support associative access in worst-case $\mathcal{O}(1)$ time with little space overhead. Consider

> **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.

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. However, this

requires a total of quadratic space. Quadratic initialization time can be avoided using the technique in [1, Exercise 2.12].

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 [33] 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:

$$f = \max_{N \text{ in dom } (\text{inv } E11)} \#(\textbf{inv } E11)\{N\} \tag{34}$$

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. For (29) and the algorithm obtained from (32), 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.

Finally, we can use linked lists when the images are small, and use hash tables when the images are larger. This achieves the same complexities analyzed in Sections 4 and 5, also for average case.

Experiments. We implemented the simplification algorithm obtained from (32) with the optimization to conditional forms and used it to replace a previous algorithm in a prototype system for dead-code analysis and elimination [25]. The prototype system is implemented using the Synthesizer Generator [36], and the simplification algorithms are written in a dialect of Scheme. We have used the system to analyze dozens of examples. Table 1 reports measurements of the most relevant parameters—as defined in the complexity analysis in Section 4, plus c4' in Section 5 and f in Section 6—and simplification times from analyzing 14 programs with 25 different queries using the new simplification algorithm.

Programs `bigfun`, `minmax`, and `biggerfun` are examples from [25]. `worst`, `worst10`, and `worst20` are examples contrived to demonstrate the worst-case cubic-time complexity. `incsort` and `incout` are incremental programs for selection sort and outer product, respectively, derived using incrementalization [27],

where dead code after incrementalization is to be eliminated. `cachebin` and `cachelcs` are dynamic-programming programs for binomial coefficients and longest common subsequences, respectively, derived using cache-and-prune [26, 24], where cached intermediate results that are not used are to be pruned. `calend`, `symbdiff`, `takr`, and `boyer` are taken from the Internet Scheme Repository [21]. `calend` is a collection of calendrical functions [10]. `takr` is a 100-function version of TAK that tries to defeat cache memory effects. `symbdiff` does symbolic differentiation. `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 1. 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.

First of all, the analysis is effective, reflected in the resulting number of live program points r compared to the total number of program points n. For some examples, the program after dead-code elimination is even asymptotically faster [25]. We also observe: 1) $\#P$ ranges from $1.02n$ to $1.56n$, 2) a is consistently very small, 3) h varies widely, 4) g and f are typically quite small, 5) $\#O$ 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.

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 1 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; both of these are slower. Optimization to conditional forms gives up to 15% speedup.

We can see that the simplification time is very much linear in c=(c1)+(c2)+(c3)+(c4), 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.

7 Related work and conclusion

Regular tree grammar based constraints have been used for analyzing recursive data in other applications and go back at least to Reynolds [38] and Schwartz [40]. Related work includes flow analysis for memory optimization by Jones and Muchnick [22], binding-time analysis for partial evaluation by Mogensen [29], set-based analysis of ML by Heintze [17], type inference by Aiken et al. [2, 3], backward slicing by Reps and Turnidge [37], and set-based analysis for debugging Scheme by Flanagan and Felleisen [13]. Some of these are general type inference and are only shown to be decidable [3] or take exponential time in the worst case [2]. For others, either a cubic time complexity is given based on a simple worst-case analysis of a relatively straightforward algorithm [17, 13], or algorithm complexity is not discussed explicitly [22, 29, 37].

Constraints have also been used for other analyses, in particular, analyses handling mainly higher-order functions or pointers. This includes higher-order

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

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

Table 1. Measurements for Example Programs.

binding-time analysis by Henglein [20], Bondorf and Jørgensen [6], and Birkedal and Welinder [4, 5], points-to analysis by Steensgaard [44], and control flow analysis for special cases by Heintze and McAllester [19]. The last restricts type sizes and has a linear time complexity, and the others use union-find algorithms [20] and have an almost linear time complexity. These analyses either do not consider recursive data structures [20, 44], or use bounded domains [6, 4, 5, 19] 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 [16] 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 [13] study techniques for component-wise simplification. Fähndrich et al. [11] study a technique for eliminating cycles in the inclusion constraint graphs. Su et al. [45] 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 [18, 25] (or quadratic-time [43]) analysis algorithm seems to give nearly linear behavior [18, 43, 25]. Our work in this paper is a start in the formal study of the reasons.

Our analysis adds edges 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 [11, 45]. However, our algorithm still proceeds in a linear fashion. That is, if we have constraints $N_1 \to N_2, ..., N_{k-1} \to N_k$, 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 [16]. For comparison, a future work would be to formalize the algorithms in [11, 45]. It will also be interesting to formalize and compare with [47]. As our problem is related to computing Datalog queries, it

will be worthwhile to see to what degree McAllester's complexity results for Datalog queries [28] could be applied; note, however, that those results are obtained based on extensive hashing and thus are for average cases, not worst cases. Compared with the magic-sets transformation [46], finite differencing or incrementalization [23] based methods derive more specialized algorithms and data structures, yielding more efficient programs, often asymptotically better.

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. Clearly, there is a large body of work on all kinds of program analysis algorithms [30], from type inference algorithms, e.g., [35], to efficient fixed-point computation, e.g., [12]. 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. [34, 32, 8, 33, 7], as illustrated in this work, and from the more formal characterization by Goyal [15].

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. 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.
3. 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.
4. 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.
5. L. Birkedal and M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3):191–208, Sept. 1995.
6. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
7. 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.
8. J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.
9. 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.
10. N. Dershowitz and E. M. Reingold. Calendrical calculations. *Software—Practice and Experience*, 20(9):899–928, Sept. 1990.
11. 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.
12. 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.
13. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, Mar. 1999.
14. F. Gecseg and M. Steinb. *Tree Automata*. Akademiai Kiado, Budapest, 1984.
15. D. Goyal. *A Language Theoretic Approach to Algorithms*. PhD thesis, Department of Computer Science, New York University, Jan. 2000.
16. 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.

17. 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.
18. 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.
19. 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.
20. 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.
21. The Internet Scheme Repository. http://www.cs.indiana.edu/scheme-repository/.
22. 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.
23. Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.
24. 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.
25. Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. In SAS 1999 [39], pages 211–231.
26. 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.
27. Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.
28. D. McAllester. On the complexity analysis of static analyses. In SAS 1999 [39], pages 312–329.
29. 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.
30. F. Nielson, H. R. Nielson, and C. Hankin, editors. *Principles of Program Analysis*. Springer-Verlag, 1999.
31. 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.
32. R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.
33. 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.
34. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.
35. J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, Apr. 1998.
36. T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1988.
37. 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.
38. 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.
39. *Proceedings of the 6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Sept. 1999.
40. J. T. Schwartz. Optimization of very high level languages – I: Value transmission and its corollaries. *Journal of Computer Languages*, 1(2):161–194, 1975.
41. 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.
42. W. K. Snyder. The SETL2 Programming Language. Technical report 490, Courant Institute of Mathematical Sciences, New York University, Sept. 1990.

43. 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.
44. B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 32–41. ACM, New York, Jan. 1996.
45. 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.
46. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, New York, 1988.
47. D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, July 1993.

A An example 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.

$$
\begin{array}{lcl}
f(x) & \stackrel{\Delta}{=} & \textbf{if } null(x) \textbf{ then } nil \textbf{ else } cons(g(car(x)), f(cdr(x)));\\
g(x) & \stackrel{\Delta}{=} & x*x*x*x*x*x*x*x;\\
len(x) & \stackrel{\Delta}{=} & \textbf{if } null(x) \textbf{ then } 0 \textbf{ else } 1 + len(cdr(x));\\
lenf(x) & \stackrel{\Delta}{=} & len(f(x));\\
\\
cons & : & cons?(car, cdr);\\
nil & : & null();
\end{array}
$$

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

$$
\begin{array}{lcl}
f(^{N_{36}}x) & \stackrel{\Delta}{=} & {}^{N_{35}}\textbf{if } {}^{N_{34}}null(^{N_{33}}x) \textbf{ then } {}^{N_{32}}nil \textbf{ else } {}^{N_{31}}cons(^{N_{30}}g(^{N_{29}}car(^{N_{28}}x)), {}^{N_{27}}f(^{N_{26}}cdr(^{N_{25}}x)));\\
g(^{N_{24}}x) & \stackrel{\Delta}{=} & {}^{N_{23}}{}^{N_{22}}{}^{N_{21}}{}^{N_{20}}{}^{N_{19}}x *^{N_{18}} x *^{N_{17}} x *^{N_{16}} x *^{N_{15}} x;\\
len(^{N_{14}}x) & \stackrel{\Delta}{=} & {}^{N_{13}}\textbf{if } {}^{N_{12}}null(^{N_{11}}x) \textbf{ then } {}^{N_{10}}0 \textbf{ else } {}^{N_9}{}^{N_8}1 +^{N_7} len(^{N_6}cdr(^{N_5}x));\\
lenf(^{N_4}x) & \stackrel{\Delta}{=} & {}^{N_3}len(^{N_2}f(^{N_1}x));
\end{array}
$$

Constructed grammar. The grammar constructed from the given program is

$$
\begin{array}{l}
N_{36} \to N_{33},\ N_{36} \to N_{28},\ N_{36} \to N_{25},\ N_{33} \to [N_{34}]cons(N_0, N_0),\ N_{33} \to [N_{34}]nil(),\ N_{34} \to [N_{35}]L,\ N_{32} \to N_{35},\\
\\
N_{28} \to [N_{29}]cons(N_{29}, N_0),\ N_{29} \to [N_{30}]N_{24},\ N_{23} \to N_{30},\ N_{30} \to car(N_{31}),\\
N_{25} \to [N_{26}]cons(N_0, N_{26}),\ N_{26} \to [N_{27}]N_{36},\ N_{35} \to N_{27},\ N_{27} \to cdr(N_{31}),\ N_{31} \to N_{35},\\
\\
N_{24} \to N_{19},\ N_{24} \to N_{18},\ N_{24} \to N_{17},\ N_{24} \to N_{16},\ N_{24} \to N_{15},\ N_{19} \to [N_{20}]L,\ N_{18} \to [N_{20}]L,\ N_{20} \to [N_{21}]L,\\
N_{17} \to [N_{21}]L,\ N_{21} \to [N_{22}]L,\ N_{16} \to [N_{22}]L,\ N_{22} \to [N_{23}]L,\ N_{15} \to [N_{23}]L,\\
\\
N_{14} \to N_{11},\ N_{14} \to N_5,\ N_{11} \to [N_{12}]cons(N_0, N_0),\ N_{11} \to [N_{12}]nil(),\ N_{12} \to [N_{13}]L,\ N_{10} \to N_{13},\\
N_8 \to [N_9]L,\ N_5 \to [N_6]cons(N_0, N_6),\ N_6 \to [N_7]N_{14},\ N_{13} \to N_7,\ N_7 \to [N_9]L,\ N_9 \to N_{13},\\
N_4 \to N_1,\ N_1 \to [N_2]N_{36},\ N_{35} \to N_2,\ N_2 \to [N_3]N_{14},\ N_{13} \to N_3,\ N_0 \to D
\end{array}
$$

User query. A user query is

$$N_3 \to L$$

Simplification result. The output of simplification, sorted by nonterminal number, is

$$
\begin{array}{ll}
N_{36} \to nil(),\ N_{36} \to cons(N_0, N_0),\ N_{36} \to cons(N_0, N_{26}), & N_{11} \to nil(),\ N_{11} \to cons(N_0, N_0),\\
N_{35} \to nil(),\ N_{35} \to cons(N_0, N_0),\ N_{35} \to cons(N_0, N_6), & N_{10} \to L,\\
N_{34} \to L, & N_9 \to L,\\
N_{33} \to nil(),\ N_{33} \to cons(N_0, N_0), & N_8 \to L,\\
N_{32} \to nil(),\ N_{32} \to cons(N_0, N_0),\ N_{32} \to cons(N_0, N_6), & N_7 \to L,\\
N_{31} \to nil(),\ N_{31} \to cons(N_0, N_0),\ N_{31} \to cons(N_0, N_6), & N_6 \to nil(),\ N_6 \to cons(N_0, N_0),\ N_6 \to cons(N_0, N_6),\\
N_{27} \to nil(),\ N_{27} \to cons(N_0, N_0),\ N_{27} \to cons(N_0, N_6), & N_5 \to cons(N_0, N_6),\\
N_{26} \to nil(),\ N_{26} \to cons(N_0, N_0),\ N_{26} \to cons(N_0, N_{26}), & N_4 \to nil(),\ N_4 \to cons(N_0, N_0),\ N_4 \to cons(N_0, N_{26}),\\
N_{25} \to cons(N_0, N_{26}), & N_3 \to L,\\
N_{14} \to nil(),\ N_{14} \to cons(N_0, N_0),\ N_{14} \to cons(N_0, N_6), & N_2 \to nil(),\ N_2 \to cons(N_0, N_0),\ N_2 \to cons(N_0, N_6),\\
N_{13} \to L, & N_1 \to nil(),\ N_1 \to cons(N_0, N_0),\ N_1 \to cons(N_0, N_{26}),\\
N_{12} \to L, & N_0 \to D
\end{array}
$$

Nonterminals N_{15} to N_{24} and N_{28} to N_{30} do not have a right-side good form. The corresponding program points are dead.