Eliminating Dead Code on Recursive Data

Yanhong A. Liu* and Scott D. Stoller*

Computer Science Department, Indiana University, Bloomington, IN 47405
{liu, stoller}@cs.indiana.edu

Abstract. This paper describes a general and powerful method for dead code analysis and elimination in the presence of recursive data constructions. We represent partially dead recursive data using liveness patterns based on general regular tree grammars extended with the notion of live and dead, and we formulate the analysis as computing liveness patterns at all program points based on program semantics. This analysis yields a most precise liveness pattern for the data at each program point, which is significantly more precise than results from previous methods. The analysis algorithm takes cubic time in terms of the size of the program in the worst case but is very efficient in practice, as shown by our prototype implementation. The analysis results are used to identify and eliminate dead code. The general framework for representing and analyzing properties of recursive data structures using general regular tree grammars applies to other analyses as well.

1 Introduction

Dead computations produce values that never get used [1]. While programmers are not likely to write code that performs dead computations, such code appears often as the result of program optimization, modification, and reuse [40, 1]. There are also other programming activities that do not explicitly involve live or dead code but rely on similar notions. Examples are program slicing [60, 45], specialization [45], incrementalization [34, 33], and compile-time garbage collection [24, 21, 42, 57]. Analysis for identifying dead code, or code having similar properties, has been studied and used widely [8, 7, 25, 41, 1, 24, 21, 10, 26, 34, 54, 45, 33, 37, 57]. It is essentially backward dependence analysis that aims to compute the minimum sufficient information needed for producing certain results. We call this dead code analysis, bearing in mind that it may be used for many other purposes.

In recent years, dead code analysis has been made more precise so as to be effective in more complicated settings [21, 10, 26, 45, 5, 33]. Since recursive data constructions are used increasingly widely in high-level languages [52, 14, 37, 3], an important problem is to identify partially dead recursive data—that is, recursive data whose dead parts form recursive substructures—and eliminate computations of them. It is difficult because recursive data structures can be defined by the user, and dead substructures may interleave with live substructures. Several methods have been studied [24, 21, 45, 33], but all have limitations.

This paper describes a general and powerful method for analyzing and eliminating dead computations in the presence of recursive data constructions. We

* The authors gratefully acknowledge the support of NSF under grant CCR-9711253 and ONR under grants N00014-99-1-0132 and N00014-99-1-0358.

1 This is different from partial dead code, which is code that is dead on some but not all computation paths [26, 5].
represent partially dead recursive data using liveness patterns based on general regular tree grammars extended with the notion of live and dead, and we formulate the analysis as computing liveness patterns at all program points based on program semantics. This analysis yields a most precise liveness pattern for the data at each program point. The analysis algorithm takes cubic time in terms of the size of the program in the worst case but is very efficient in practice, as shown in our prototype implementation. The analysis results are used to identify and eliminate dead code. The framework for representing and analyzing properties of recursive data structures using general regular tree grammars applies to other analyses as well.

The rest of the paper is organized as follows. Section 2 describes a programming language with recursive data constructions. Section 3 defines liveness patterns that represent partially dead recursive data. Section 4 formulates the analysis as solving sets of constraints on grammars. Section 5 presents efficient algorithms for computing liveness patterns at every program point. Section 6 describes dead code elimination, our implementation, and extensions. Section 7 compares this work with related work and concludes.

2 Language

We use a simple first-order functional programming language. The expressions of the language are:

\[
e ::= v \quad \text{variable} \\
c(c_1, \ldots, c_n) \quad \text{constructor application} \\
p(c_1, \ldots, c_n) \quad \text{primitive function application} \\
\text{if } c_1 \text{ then } e_2 \text{ else } e_3 \quad \text{conditional expression} \\
\text{let } v = e_1 \text{ in } e_2 \quad \text{binding expression} \\
f(c_1, \ldots, c_n) \quad \text{function application}
\]

Each constructor \( c \), primitive function \( p \), and user-defined function \( f \) has a fixed arity. If a constructor \( c \) has arity 0, then we write \( c() \) instead of \( c() \). New constructors can be declared, together with their arities. When needed, we use \( c^n \) to denote that \( c \) has arity \( n \). For each constructor \( c^n \), there is a primitive function \( c \) that tests whether the argument is an application of \( c \), and if \( n > 0 \), then for each \( i = 1..n \), there is a primitive function \( c_i \) that selects the \( i \)th component in an application of \( c \), e.g., \( c_2(c^3(x, y, z)) = y \). A program is a set of mutually recursive function definitions of the form:

\[
f(v_1, \ldots, v_n) \triangleq e
\]

together with a set of constructor declarations. Figure 1 gives some example definitions, assuming that \( \text{min} \) and \( \text{max} \) are primitive functions, and that constructors \( \text{nil}^0 \), \( \text{cons}^2 \), and \( \text{triple}^3 \) are declared in the programs where they are used. For ease of reading, we use \( \text{null} \) instead of \( \text{nil}^? \), \( \text{car} \) instead of \( \text{cons}^? \), \( \text{cdr} \) instead of \( \text{cons}^? \), and \( 1\text{st} \), \( 2\text{nd} \), and \( 3\text{rd} \) instead of \( \text{triple}^? \), \( \text{triple}^? \), and \( \text{triple}^? \), respectively.

This language has call-by-value semantics. Well-defined expressions evaluate to constructed data, such as \( \text{cons}(3, \text{nil}) \). We use \( \bot \) to denote the value of undefined (non-terminating) expressions; an expression must evaluate to \( \bot \) if any of its subexpressions evaluates to \( \bot \). Since a program can use data constructions
\begin{align*}
\text{min-max}(x) & : \text{compute min and max for all suffixes of } x \\
\text{min-max}(x) & \triangleq \text{if null}(x) \text{ then nil} \\
& \quad \text{else if null(cdr(x)) \text{ then}} \\
& \quad \text{cond}(\text{triples(car(x)), car(x), car(x)), \text{nil}) \\
& \quad \text{else let } v = \text{min-max(cdr(x)) \text{ in}} \\
& \quad \text{cond}(\text{triples(car(x)),} \\
& \quad \quad \text{min(car(x), cdr(car(x)))}, \\
& \quad \quad \text{max(car(x), cdr(car(x)))},) \\
& \quad \text{v} \\
\text{list-second}(x) & : \text{list the second element in each triple in } x \\
\text{list-second}(x) & \triangleq \text{if null}(x) \text{ then nil} \\
& \quad \text{else cond}(2\text{nd(car(x)), list-second(cdr(x)))} \\
\text{get-min}(x) & : \text{compute the min elements for all suffixes of } x \\
\text{get-min}(x) & \triangleq \text{list-second(min-max(x))} \\
\text{len}(x) & : \text{compute length of } x \\
\text{len}(x) & \triangleq \text{if null}(x) \text{ then } 0 \\
& \quad \text{else } 1 + \text{len(cdr(x))} \\
\text{odd}(x) & : \text{get elements of } x \text{ at odd positions} \\
\text{even}(x) & : \text{get elements of } x \text{ at even positions} \\
\text{odd}(x) & \triangleq \text{if null}(x) \text{ then nil} \\
& \quad \text{else cond(car(x),} \\
& \quad \quad \text{even(cdr(x)))} \\
\text{even}(x) & \triangleq \text{if null}(x) \text{ then nil} \\
& \quad \text{else odd(cdr(x))}
\end{align*}

\textbf{Fig. 1. Example function definitions.}

c(e_1, \ldots, e_n) in recursive function definitions, it can build data structures of
unbounded sizes, i.e., sizes not bounded in any way by the size of the program but
determined by particular inputs to the program.

There can be values, which can be subparts of constructed data, computed
by a program that are not needed in obtaining the output of the program. To
improve program efficiency, we can eliminate such dead computations and use a
special symbol \_ as a placeholder for their values. A constructor application does
not evaluate to \_ even if some arguments evaluate to \_. A primitive function
application (or a conditional expression) must evaluate to \_, if not \_, if any
of its subexpressions (or the condition, respectively) evaluates to \_. Whether
a function application (or a binding expression) evaluates to \_ depends on the
values of the arguments (or the bound variable, respectively) and how they are
used in the function definition (or the body, respectively).

Dead code may exist in a program especially when only certain parts of its
result or intermediate results are needed. Such parts can be specified by a user
or determined by how these results are used in computing other values, e.g., by
how the value of \text{min-max} is used in computing \text{get-min}, in which case all the
max operations are dead.

3 Liveness patterns

We represent partially dead recursive data using liveness patterns. A liveness
pattern indicates which parts of data must be dead and which parts may be live.
\text{D} indicates that a part must be completely dead, and \text{L} indicates that a part
may be completely live. Partial liveness is represented using constructors. For
example, \text{cons(D, L)} indicates a \text{cons} structure with a definitely dead head and
a possibly live tail. Also, \text{nil()} indicates the liveness pattern corresponding to
a \text{nil} structure, so there is no confusion between a liveness pattern and a data
value. A liveness pattern is a function; when applied to a data value, it returns
the data with the live parts unchanged and the dead parts replaced by \_. For example,
\begin{align*}
\text{cons}(\text{D, cons(L, D)}) (\text{cons(0, cons(1, cons(2, nil())))}) = \text{cons(_, cons(1, _))}.
\end{align*}
Formally, liveness patterns are domain projections [48, 17], which provide a clean tool for describing substructures of constructed data by projecting out the parts that are of interest [56, 27, 39, 45, 33]. Let $X$ be the domain of all possible values computed by our programs, including $\perp$ and values containing $\_$. We define an ordering $\sqsubseteq$ on $X$, where we read $x_1 \sqsubseteq x_2$ as "$x_1$ is more dead than $x_2$": for all $x$ in $X$, $\perp \sqsubseteq x$, and for two values $x_1$ and $x_2$ other than $\perp$,

$$x_1 \sqsubseteq x_2 \text{ iff } x_1 = \_ \text{, or } x_1 = x_2 \text, \text{ or }$$

$$x_1 = c(x_{11}, \ldots, x_{1n}), \quad x_2 = c(x_{21}, \ldots, x_{2n}), \text{ and } x_{1i} \sqsubseteq x_{2i} \text{ for } i = 1, \ldots, n.$$  \hspace{1cm} (2)

A liveness pattern over $X$ is a function $\pi : X \rightarrow X$ such that $\pi(x) \sqsubseteq x$ and $\pi(\pi(x)) = \pi(x)$ for all $x \in X$. $L$ is the identity function: $L(x) = x$, $D$ is the absence function: $D(x) = \_ \text{ for all } x \neq \perp$, and $D(\perp) = \perp$. $c^n(\pi_1, \ldots, \pi_n)$ is the function:

$$c^n(\pi_1, \ldots, \pi_n)(x) = \begin{cases} 
  c^n(\pi_1(x_1), \ldots, \pi_n(x_n)) \text{ if } x = c^n(x_1, \ldots, x_n) \\
  \_ \text{ otherwise}
\end{cases}$$ \hspace{1cm} (3)

Grammar-based liveness patterns. We represent liveness patterns as grammars. For example, the grammar $S \rightarrow \text{nil()} | \text{cons}(D,S)$ projects out a list whose elements are dead but whose spine is live. It generates the set of sentences $\{\text{nil()}, \text{cons}(D,\text{nil()})], \text{cons}(D,\text{cons}(D,\text{nil()})], \ldots\}$. Applying each element to a given value, say, $\text{cons}(2, \text{cons}(4, \text{nil}))$, yields $\perp, \perp, \text{cons}(\_, \text{cons}(\_, \text{nil})), \perp, \ldots$, in which $\text{cons}(\_, \text{cons}(\_, \text{nil}))$ is the least upper bound.

Formally, the grammars we use for describing liveness patterns are regular tree grammars [16], which allow bounded, and often precise, representations of unbounded data [23, 38, 39, 2, 51, 9, 45]. A regular-tree-grammar-based liveness pattern $G$ is a quadruple $(T, N, \mathcal{P}, S)$, where $T$ is a set of terminal symbols including $L, D$, and all possible constructors $c, N$ is a set of nonterminal symbols $\mathcal{N}$ is a set of productions of the form:

$$N \rightarrow D, \quad N \rightarrow L, \quad \text{or } N \rightarrow c^n(N_1, \ldots, N_n),$$  \hspace{1cm} (4)

and nonterminal $S$ is the start symbol. So, our liveness patterns use general regular tree grammars [23, 16, 9] extended with the special constants $D$ and $L$. The language $L_G$ generated by $G$ is the set $\{\pi \in T^* | S \rightarrow_G^* \pi\}$ of sentences. The projection function that $G$ represents is:

$$G(x) = \sqcup\{\pi(x) \mid \pi \in L_G\}$$ \hspace{1cm} (5)

where $\sqcup$ is the least upper bound operation for $\sqsubseteq$. It is easy to see that $G(x)$ is well-defined for all $x \in X$. We overload $L$ to denote the grammar that generates sentence $L$, and overload $D$ to denote the grammar that generates only sentence $D$. For ease of presentation, when no confusion arises, we write grammars in compact forms. For example, $\{S \rightarrow \text{nil()} | \text{cons}(N,S), N \rightarrow \text{triple}(D,L,D)\}$, where $\mid$ denotes alternation, projects out a list whose elements are triples whose first and third components are dead.

We extend regular tree grammars to allow productions of the form:

$$N \rightarrow N', \quad N \rightarrow c_1(N'), \quad \text{or } N \rightarrow [N'] R'$$  \hspace{1cm} (6)
for $R'$ of the form $L, c^n(N_1, \ldots, N_n)$, or $N'$, and we define:

$$c_T(\pi) = \begin{cases} L & \text{if } \pi = L \\ \pi_i & \text{if } \pi = c^n(\pi_1, \ldots, \pi_n) \\ D & \text{otherwise} \end{cases}$$

and $[\pi]^{\pi'} = \begin{cases} D & \text{if } \pi = D \\ \pi' & \text{otherwise} \end{cases}$ (7)

These extended forms are for convenience later; the selector form in the middle of (6) is the same as that first used by Jones and Muchnick [23], and the conditional form on the right of (6) is for similar purposes as those used in several other analyses, e.g., the operator $\triangleright$ used by Wadler and Hughes for strictness analysis [56]. Given an extended regular tree grammar $G$ that contains productions of the forms in (4) and (6), we can construct a regular tree grammar $G'$ that contains only productions of the form (4) such that $L_G = L_{G'}$, i.e., $G'$ and $G$ represent the same projection function; an algorithm is given in Section 5.

When using a grammar $G$, what matters is the projection function that $G$ represents. In fact, different grammars can represent the same projection function. A basic idea of this work is to capture the information of interest—liveness patterns—using grammars that are constructed based on program semantics and then simplify the grammars to equivalent grammars in simpler forms where, in particular, the only grammar that represents $D$ is $\{S \rightarrow D\}$.

We define an ordering $\leq$ on regular-tree-grammar-based liveness patterns. For two grammars $G_1$ and $G_2$, we define:

$$G_1 \leq G_2 \text{ iff } \forall \pi_1 \in L_{G_1}, \exists \pi_2 \in L_{G_2}, \pi_1 \leq \pi_2,$$ (8)

where for two sentences $\pi_1$ and $\pi_2$, we define (overloading $\leq$):

$$\pi_1 \leq \pi_2 \text{ iff } \pi_1 = D, \text{ or } \pi_2 = L, \text{ or } \pi_1 = c(\pi_{11}, \ldots, \pi_{1n}), \pi_2 = c(\pi_{21}, \ldots, \pi_{2n}), \text{ and } \pi_{1i} \leq \pi_{2i} \text{ for } i = 1..n.$$ (9)

For convenience, we define $G_1 \geq G_2$ if and only if $G_2 \leq G_1$. It is easy to see that

$$\text{if } G_1 \leq G_2, \text{ then } \forall x, G_1(x) \subseteq G_2(x).$$ (10)

This means that if $G_1 \leq G_2$, then $G_1$ projects out values that are more dead than those that $G_2$ projects out. This is a basis of our correctness proof. The converse is not true, e.g., $G_1 = \{S \rightarrow \text{cons}(L, L)\}$ and $G_2 = \{S \rightarrow \text{cons}(L, D) | \text{cons}(D, L)\}$ form a counterexample for the converse, but this converse is not used. Note that this ordering on grammars does not form a complete partial order.

**Notation.** We use $\text{CON}$ to denote the grammar that projects any constructor but none of its arguments: letting $T_c$ be the set of all possible constructors,

$$\text{CON} = \langle T_c \cup \{D\}, \{S\}, \{S \rightarrow c^n(D_1, \ldots, D_n) \mid c^n \in T_c\}, S \rangle.$$ 2 (11)

Given a grammar $G = \langle T, N', P, S \rangle$, we use $\text{con}_{c,i}(G)$ to denote using $G$ as the $i$th component of a $c^n$ structure $(i \leq n)$ whose other components are dead: assuming $S'$ is a nonterminal not used in $G$,

$$\text{con}_{c,i}(G) = \langle T \cup N' \cup \{S'\}, P \cup \{S' \rightarrow c^n(D_1, \ldots, D_n, S, D, \ldots, D)\}, S' \rangle.$$ 2 (12)
and we use \( sel_{c,i}(G) \) to denote the part of \( G \) corresponding to the \( i \)th component of a \( c^n \) structure (\( i \leq n \)): assuming \( S' \) is a nonterminal not in used in \( G \),

\[
sel_{c,i}(G) = \{ T, N \cup \{ S' \}, P \cup \{ S' \rightarrow c_i(S) \}, S' \}. \tag{13}
\]

For example, if \( \text{nil} \) and \( \text{cons} \) are all possible constructors where a liveness pattern \( CON \) is used, as we assume for functions \( \text{len}, \text{odd}, \) and \( \text{even} \), then \( CON = \{ S \rightarrow \text{nil}(\text{cons}(D, D)) \} \). If \( G = \{ S \rightarrow L \} \), then \( con_{\text{cons},1}(G) = \{ S' \rightarrow \text{cons}(S, D), S \rightarrow L \} \) and \( sel_{\text{cons},1}(G) = \{ S' \rightarrow \text{car}(S), S \rightarrow L \} \). Finally, we define a conditional:

\[
\text{cond}(G_1, G_2) = \begin{cases} D & \text{if } G_1 = D \\ G_2 & \text{otherwise.} \end{cases} \tag{14}
\]

4 Analysis of liveness patterns using constraints

Dead code analysis computes liveness patterns associated with values at program points, such as function definitions, parameters, and subexpressions. We develop such a backward dependence analysis. Given liveness patterns associated with certain program points, it computes liveness patterns at all program points, so that the liveness specified by the given liveness patterns is guaranteed. The basic idea is that a liveness pattern associated with a program point is constrained by liveness patterns associated with other points based on the semantics of the program segments involved.

Sufficiency conditions. The resulting liveness patterns must satisfy two kinds of sufficiency conditions. First, the resulting grammar at a program point must project out values that are more live than required by the given grammar (if any) associated with that point. Precisely, at each subexpression \( e \) where a liveness pattern is given, if the given grammar is \( G_0 \), and the resulting grammar is \( G \), then \( G_0(e) \subseteq G(e) \) for all values of the free variables in \( e \). Second, the resulting grammars must satisfy the constraints determined by the program semantics. Precisely, assume a resulting grammar is associated with each parameter and each subexpression of all function definitions. Let \( G_e \) denote that grammar \( G \) is associated with \( e \). Then (1) the liveness patterns at function parameters must be sufficient to guarantee the liveness pattern at the function return, i.e., for each definition of the form \( f(G_1(v_1), \ldots, G_n(v_n)) \triangleq G_e \), the following sufficiency condition must be satisfied for all values \( v_1, \ldots, v_n \):

\[
G(f(v_1, \ldots, v_n)) \subseteq f(G_1(v_1), \ldots, G_n(v_n)) \tag{15}
\]

and (2) the liveness pattern at each subexpression must be sufficient to guarantee the liveness pattern at the enclosing expression, i.e., for each subexpression that is of a form in the left column below, the corresponding sufficiency condition in the right column must be satisfied for all values of the free variables in the
subexpression:

\[G_{\varepsilon_{1},...,\varepsilon_{n}}(e_{1},...,e_{n})\]

\[G(\varepsilon_{1},...,\varepsilon_{n}) \subseteq c(G_{1}(e_{1}),...,G_{n}(e_{n}))\]

\[G(\varepsilon' c_{1}) \subseteq c(G_{1}(e' c_{1}))\]

\[G(\varepsilon' c_{2}) \subseteq c(G_{1}(e' c_{2}))\]

\[G(q(\varepsilon_{1},...,\varepsilon_{n})) \subseteq q(G_{1}(\varepsilon_{1}),...,G_{n}(\varepsilon_{n}))\]

\[G(\text{let } y = e_{1} \text{ in } e_{2}) \subseteq \text{let } y = G_{1}(e_{1}) \text{ in } G_{2}(e_{2})\]

\[G(f(G_{1},...,G_{n})) \subseteq f(G_{1}(e_{1}),...,G_{n}(e_{n}))\]

Note that no approximation is made in these conditions. For example, the condition of a conditional expression does not have to be evaluated, so it does not have to be associated with \(L\). In particular, the liveness patterns associated with a function application are not related to liveness patterns associated with the definition of the function, and thus, different applications of the same function may require different parts of the function definition to be live. For example, consider functions \(f\) and \(g\) below:

\[f(x, y) \triangleq \text{if } x > 0 \text{ then } x \text{ else } y\]

\[g(z) \triangleq G_{0} : f(x, y) + g(x, z)\]

Given \(G_{0} = L\) at the definition of \(g\), the liveness patterns associated with all program points, where the liveness patterns not explicitly written are all \(L\), satisfy the sufficiency conditions. Note that the two calls to \(f\) need different parts of \(f\) to be live.

Grammar constraints. Given liveness patterns associated with certain subexpressions, we construct a set of constraints on the resulting liveness patterns that guarantee the sufficiency conditions. First, at each subexpression \(e\) where a liveness pattern is given, if the given grammar is \(G_{0}\), and the resulting grammar is \(G\), then we construct \(G \geq G_{0}\). Second, for a function definition of the form \(f(G_{1},...,G_{n}) \triangleq e\), we construct, for \(i = 1..n\) and for each occurrence \(G_{i}:v_{i}\) in \(e\), the constraint

\[G_{i} \geq G'_{i}\]  \hspace{1cm} (16)

and, for each subexpression \(e\) that is of a form in the left column of Figure 2, we construct the corresponding constraints in the right column. These constraints make approximations while guaranteeing the sufficiency conditions, as explained below.

Formula (16) for function definitions requires that the liveness pattern at formal parameter \(v_{i}\) be greater than or equal to the liveness patterns at all uses of \(v_{i}\). Rule (7) for function calls requires that, for all non-dead calls of the same function, the liveness patterns at the arguments be greater than or equal to the liveness patterns at the corresponding formal parameters, and that the liveness pattern for the return value of the function be greater than or equal to the liveness patterns at all calls. Thus, if a function call is dead, then all its arguments are also dead, even though the formal parameters of the function may not be dead due to other calls to the same function.

Other constraints are based on the semantics of each construct locally. Rules (1)-(3) handle data constructions. Rule (1) says that liveness pattern at a component of a construction must be no less than the corresponding component in
the liveness pattern at the result of the construction. As a special case of (1), for any constructor of arity 0, no constraint is added. Rule (2) requires that, if the result of a selection by \( c_i \) is not dead, then the argument be as live as a constructor using \( c \) whose \( i \)th component is as live as the result of the selection. Rule (3) says that, if the result of an application of a tester is not dead, then the liveness pattern at the argument needs to project out the outermost constructor but none of the components. Rule (4) says that, if the result of a primitive operation is not dead, then each argument must be live. If we assume that primitive functions are defined only on primitive types such as Boolean and integer, then we could use \( CON \) in place of \( L \) in the constraints. Rule (5) requires that the condition be live if the result of the entire conditional expression is not dead, and that both branches be as live as the result. Again, we could use \( CON \) in place of \( L \) as a sufficient context for \( e_1 \); furthermore, if \( e_2 \) equals \( e_3 \); in fact as long as \( G(e_2) \) equals \( G(e_3) \), then we could use \( D \) in place of \( L \) as a sufficient context for \( e_1 \) and thus no constraint for \( G_1 \) would be needed. Rule (6) is similar to a function call, since it equals an application of \( \lambda u. e_2 \) to \( e_1 \). It requires that the defining expression \( e_1 \) be as live as all its uses in the body, and that the body be as live as the result.

We can show by a standard inductive argument that the constraints for each construct guarantee sufficient information, and thus an inductive proof shows that the sufficiency conditions are satisfied.

5 Construction and simplification of liveness patterns

We describe a straightforward method for building minimum grammars that satisfy the above constraints; these grammars may contain productions of the extended forms in (6). Then, we simplify the grammars by eliminating extended forms; this makes explicit whether the grammar associated with a program point equals dead.

\textbf{Constructing the grammars.} Let \( T_n \) be the set of all possible constructors in the program. Let \( N_0 \) be the set of nonterminals used in the given liveness patterns associated with selected subexpressions. We associate a unique nonterminal, not in \( N_0 \), with each parameter and each subexpression of all function definitions.
Then we add productions using these terminals and nonterminals. Finally, the resulting grammar at a program point is formed by using these terminals, nonterminals, and productions, and by using the nonterminal associated with that point as the start symbol.

We add two kinds of productions. For each subexpression $e$ where a grammar $G_0$ is given, let $N_0$ be the start symbol of $G_0$, and let $N$ be the nonterminal associated with $e$. We add $N \rightarrow N_0$ as well as all productions in $G_0$. Second, for each function definition $f(N_1; \ldots; N_n \leftarrow e)$, we add, for each $i = 1..n$ and for each occurrence $N_i^p$ in $e$, the production

$$N_i \rightarrow N_i'$$  \hfill (17)  

and, for each subexpression of $e$ that is of a form in the left column of Figure 3, the corresponding productions in the right column.

---

(1) $N_i(c(N_1; \ldots; N_n\leftarrow e_n))$ \hfill $N_i \rightarrow c(N)$ for $i = 1..n$

(2) $N_i(c(N_1; \ldots; N_n))$ \hfill $N' \rightarrow [N]c((D_1; \ldots; D_i; N; D_i; \ldots; D))$

\hfill with $i-1$ $D_i$'s before $N$ and $n-i$ $D_i$'s after $N$

(3) $N_i(c(N_1; \ldots; N_n))$ \hfill $N' \rightarrow [N]c((D_1; \ldots; D))$

\hfill with $n$ $D_i$'s for each possible constructor $c$

(4) $N_i[q(N_1; \ldots; N_n \leftarrow e_n)$ if $q \neq p$ other than $c^r$ or $c^l$

\hfill $N_i \rightarrow [N]L$ for $i = 1..n$

(5) $N_i$ if $N_1 \leftarrow N_2$ else $N_3$ \hfill $N_1 \rightarrow [N]L$, $N_2 \rightarrow N$, $N_3 \rightarrow N$

(6) $N_i$ let $u = N_1 \leftarrow N_2$ \hfill $N_i \rightarrow [N]N'$ for each occurrence of $N_i' \leftarrow e$ in $e_n$, $N_2 \rightarrow N$

(7) $N_i[f(N_1; \ldots; N_n \leftarrow e_n)$ where $f(N_1; \ldots; N_n \leftarrow e_n) = N_i' \leftarrow e$

\hfill $N_i \rightarrow [N]N'$ for $i = 1..n$, $N' \rightarrow N$

---

It is easy to show that the resulting grammars satisfy the grammar constraints in Figure 2 and thus give sufficient information at every program point. To show this, simply notice that the productions in Figure 3 can be obtained from the constraints in Figure 2 by replacing $G$ with $N$ and $\geq$ with $\rightarrow$, and by replacing grammar operations with the corresponding productions based on definitions: $\text{sel}_c(G)$ with $c(N)$, $\text{com}_c(G)$ with $c(D_1; \ldots; D; N; D; \ldots; D)$, $\text{CON}$ with $c(D_1; \ldots; D)$ for all $c$, and $\text{cond}(G_1; G_2)$ with $[N_1]N_2$. Thus, each production constructed here guarantees exactly a corresponding grammar constraint in Figure 2 simply by definitions. Furthermore, the resulting grammars are minimal among all solutions that use the same set of nonterminals, and they give minimum sufficient information. To see this, notice that a smaller grammar at any point would make the nonterminal at that point correspond to a smaller grammar than the grammar generated by the right hand side(s) of the nonterminal, violating the corresponding grammar constraints.

Let $n$ denote the size of the program. Assume that the maximum arity of constructors, primitive functions, and user-defined functions is bounded by a constant. Since a constant number of productions are added at each program point, the above construction takes $O(n)$ time.
**Example 1.** For functions \( \text{len} \), \( \text{odd} \), and \( \text{even} \) in Figure 1, the nonterminals labeling the program points and the added productions are shown in Figure 4. For example, we have \( N_{12} \rightarrow [N_{13}] \text{cons}(N_{13}, N_0) \), where \( N_0 \rightarrow D \) is the last production on the last line. It means that \( N_{12} \) is conditioned on \( N_{13} \): if \( N_{13} \) is \( D \), so is \( N_{12} \), otherwise \( N_{12} \) is \( \text{cons}(N_{13}, N_0) \), i.e., it projects out a \( \text{cons} \) structure, the first component of which is projected out by \( N_{13} \). Suppose we need the result of \( \text{len} \); we add \( N_{28} \rightarrow L \), since \( N_{28} \) corresponds to the return value of \( \text{len} \). Suppose we need the result of \( \text{odd} \); we add \( N_{18} \rightarrow L \). Suppose we need to know whether the result of \( \text{odd} \) is \( \text{nil} \) or \( \text{cons} \); we add \( N_{18} \rightarrow \text{nil}() \), \( N_{18} \rightarrow \text{cons}(D, D) \).

**Simplifying the grammars.** The grammars obtained above may contain productions of the extended forms in (6) and thus be difficult to understand and use. We simplify the grammars by removing extended forms using an iterative algorithm given in Figure 5. After the simplification, nonterminals that do not appear on the left side of a production with \( L \) or \( c(N_1, ..., N_n) \) on the right side are implied.
to derive only $D$. We can read off the grammar at any function parameter or subexpression by starting at the associated nonterminal and collecting all productions whose left sides are reachable from this start symbol. The correctness of this algorithm is based on the definitions of the extended forms and can be proved in a similar way to when only the selector form is used [23].

Nonterminals are associated with program points, so there are $O(n)$ of them. Each step adds a production of the form $N 	o L$, $N 	o c(N_1, \ldots, N_k)$, or $N 	o N'$. Since each right side of the form $c(N_1, \ldots, N_k)$ is among the right sides of the originally constructed grammar, there are at most $O(n)$ of them. Thus, for each nonterminal, at most $O(n)$ productions are added. So totally at most $O(n^2)$ productions are added. Adding a production has $O(n)$ cost to check what other productions to add. Thus, the overall simplification takes $O(n^3)$ time. Although this appears expensive, the analysis is very fast in practice, as shown by our prototype implementation.

**Example 2.** Suppose we need the result of $\text{len}$ and therefore added $N_{28} \to L$; we obtain the productions

$$N_{29} \to \text{nil}(), N_{28} \to \text{cons}(N_0, N_0), N_{29} \to \text{cons}(N_0, N_{21}), N_{28} \to L, N_{27} \to L, N_{25} \to \text{nil}().$$

Suppose we need the result of $\text{odd}$ and therefore added $N_{18} \to L$; we obtain the productions

$$N_{19} \to \text{cons}(N_0, N_0), N_{19} \to \text{nil}(), N_{18} \to \text{cons}(N_{12}, N_0), N_{19} \to \text{cons}(N_0, N_{18}), N_{18} \to L,$$

In each case, other nonterminals derive only $D$.

The resulting grammars can be further simplified by minimization [16], but minimization is not needed for identifying dead code, since minimization does not affect whether a nonterminal derives only $D$.

### 6 Dead code elimination, implementation, and extensions

Consider all function parameters and subexpressions whose associated liveness patterns are $D$. These parts of the program are dead, so we eliminate them by replacing them with $\_\_$. In 

**Example 3.** Suppose we need to know whether the result of $\text{odd}$ is $\text{nil}$ or $\text{cons}$ and therefore added $N_{18} \to \text{nil}()$, $N_{18} \to \text{cons}(D, D)$; eliminating dead code based on the simplified grammar in Example 2, where $N_1$ to $N_{13}$ all have only $D$ on the right hand sides, yields:

$$\text{odd}(x) \triangleq \begin{cases} \text{null}(x) \text{ then } \text{nil} & \\ \text{else } \text{cons}(\_, \_) & \end{cases} \quad (18)$$
Suppose we need the result of function \textit{getmin}, given in Figure 1; analyzing and eliminating dead code yields the following function along with functions \textit{listsecond} and \textit{getmin}:

\[
\text{minmax}(x) \triangleq \begin{cases} 
\text{if } \text{null}(x) \text{ then } \text{nil} \\
\text{else if } \text{null}(	ext{cdr}(x)) \text{ then } \text{cons}\{\text{car}(x), \text{nil}\} \\
\text{else let } v = \text{minmax}(	ext{cdr}(x)) \text{ in } \text{cons}\{\text{car}(x), \text{min}(\text{car}(x), 2\text{nd}(\text{car}(v))), v\} 
\end{cases}
\]

(19)

As another example, if the result of \textit{minmax} is used as argument to \textit{len} instead of \textit{listsecond}, then our algorithm finds that the entire \textit{triple} constructions in \textit{minmax} are dead. However, if the result of \textit{minmax} is used as argument to \textit{odd}, then none of the subexpressions in \textit{minmax} is dead, since the triple is used in every odd recursive call.

Our dead code elimination preserves semantics in the sense that, if the original program terminates with a value, then the new program terminates with the same value.

Two further optimizations are possible but need further study. First, \textit{minmax} in (19) can be further optimized by removing the \textit{triple} constructions and selectors. Second, when the result of \textit{minmax} is used as argument to \textit{odd}, there is no dead code in \textit{minmax}, but the triple in every even call is indeed dead. One needs to unfold the definition of \textit{minmax} to remove such \textit{dead computations}.

Eliminating dead code may improve efficiency in many ways. First, the resulting programs can run faster and use less space. Additionally, compilation of the optimized programs takes less time and also less space, which is especially desirable when using libraries. Furthermore, smaller programs are easier to understand and maintain, yielding higher software productivity.

\textit{Implementation}. We have implemented the analysis in a prototype system. The implementation uses the Synthesizer Generator [44]. The algorithm for simplifying the grammars is written in the Synthesizer Generator Scripting Language, STk, a dialect of Scheme, and consists of about 300 lines of code. Other parts of the system support editing of programs, display of nonterminals at program points, construction of grammars, highlighting of dead code, etc., and consist of about 3000 lines of SSL, the Synthesizer Generator Specification Language. All the grammars for the examples in this paper are generated automatically using the system.

We have used the system to analyze dozens of examples. The lengths of those programs range from dozens of lines to over a thousand lines. The analysis, although written in STk, is very efficient. Our original motivation for studying this general problem was for identifying appropriate intermediate results to cache and use for incremental computation [33]. There, we propose a method, called cache-and-prune, that first transforms a program to cache all intermediate results, then reuses them in a computation on incremented input, and finally prunes out cached values that are not used. Reusing cached values often produces asymptotic speedup, but leaving in unused values can be extremely inefficient. The analysis method studied in this paper, when adopted for pruning, is extremely effective. The pruned programs consistently run faster, use less space, and are smaller in code size. We also used the analysis for eliminating dead code in deriving incremental programs [34]. There, the speedup is often asymptotic. For example, dead code elimination enables incremental selection sort to improve from \(O(n^2)\) time to \(O(n)\) time.
Figure 6 summarizes the experimental results for a number of examples. Program mimmax is as in Figure 1. Programs incsort and incout [34] are derived incremental programs for selection sort and outer product, respectively, where dead code after incrementalization is to be eliminated. Programs cachebin and cachels [31] are dynamic-programming programs transformed from straight-forward exponential-time programs for computing binomial coefficients and longest common subsequences, respectively, with intermediate results cached, reused, and to be pruned. Program calend is a collection of calendrical calculation functions [12], and program takr is a 100-function version of TAK that tries to defeat cache memory effects [47].

<table>
<thead>
<tr>
<th>program name</th>
<th>functions of interest</th>
<th>total initial points</th>
<th>dead program points</th>
<th>live program points</th>
<th>number of initial productions</th>
<th>number of resulting productions</th>
<th>analysis time (ms) incl GC</th>
<th>analysis time (ms) excl GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>mimmax</td>
<td>getlen</td>
<td>84</td>
<td>40</td>
<td>44</td>
<td>111</td>
<td>91</td>
<td>0.019</td>
<td>0.016</td>
</tr>
<tr>
<td>mimmax</td>
<td>getmin</td>
<td>81</td>
<td>32</td>
<td>49</td>
<td>111</td>
<td>91</td>
<td>0.017</td>
<td>0.011</td>
</tr>
<tr>
<td>incsort</td>
<td>sort</td>
<td>108</td>
<td>81</td>
<td>27</td>
<td>144</td>
<td>34</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>incout</td>
<td>out</td>
<td>117</td>
<td>62</td>
<td>55</td>
<td>144</td>
<td>78</td>
<td>0.007</td>
<td>0.006</td>
</tr>
<tr>
<td>calend</td>
<td>calendar absolute</td>
<td>1551</td>
<td>1309</td>
<td>128</td>
<td>1839</td>
<td>272</td>
<td>0.017</td>
<td>0.008</td>
</tr>
<tr>
<td>calend</td>
<td>islamic date</td>
<td>1551</td>
<td>1205</td>
<td>346</td>
<td>1839</td>
<td>419</td>
<td>0.083</td>
<td>0.069</td>
</tr>
<tr>
<td>calend</td>
<td>eastern orthodox date</td>
<td>1551</td>
<td>1176</td>
<td>375</td>
<td>1839</td>
<td>461</td>
<td>0.086</td>
<td>0.069</td>
</tr>
<tr>
<td>calend</td>
<td>yahrezeit</td>
<td>1551</td>
<td>1123</td>
<td>428</td>
<td>1839</td>
<td>4851</td>
<td>0.086</td>
<td>0.069</td>
</tr>
<tr>
<td>takr</td>
<td>run-takr</td>
<td>2804</td>
<td>0</td>
<td>2804</td>
<td>4005</td>
<td>2805</td>
<td>0.303</td>
<td>0.304</td>
</tr>
<tr>
<td>takr</td>
<td>tak99</td>
<td>2804</td>
<td>4</td>
<td>2800</td>
<td>4005</td>
<td>2801</td>
<td>0.419</td>
<td>0.310</td>
</tr>
</tbody>
</table>

*Fig. 6.* Experimental results.

When using dead code analysis for incrementalization and for pruning unused intermediate results, there is always a particular function of interest, shown in Figure 6. For general programs, especially libraries, such as the calend example, there may not be a single function that is of interest, so we have applied the analysis on several different functions of interest.

The size of a program is precisely captured by the total number of program points, which for most programs is about twice the number of lines of code. The number of dead program points depends on both the program and the function of interest. For example, for libraries, such as the calend program, much dead code is found, whereas for takr, all 100 functions other than the driver function run-takr, are involved in calling each other. Our highlighting allows us to easily see the resulting live or dead slices. For example, for several functions in the calend program, only the slice for date, not year or month, is needed. We can see the number of initial productions is roughly linear in the size of the given program, and the number of resulting productions is roughly linear in the number of live program points.

The analysis time for simplifying the grammars, in milliseconds, is measured on an Ultra 10 with 299MHz CPU and 124 MB main memory. We can see that the analysis time is roughly linear in the number of live program points. This is important, especially for analyzing libraries, where being linear in the size of the entire program is clearly not good. We achieved this high efficiency by a careful but simple optimization in our simplification algorithm: after adding a new production, we consider only productions in extended forms whose right-hand sides use the left-hand side symbol of the new production. This makes the
analysis proceed in an incremental fashion, and only program points that are not dead are followed.

To summarize, our method produces precise analysis results as desired. The analysis is also very fast compared with other reported analyses using constraints. For example, Heintze’s analysis takes on the order of seconds for programs of 100 lines to over 1000 lines [19].

Figure 7 is a screen dump of the system on a small example of four functions and constructors nil and cons. Program points are annotated with nonterminals highlighted in red. The shaded region contains the function of interest. The two sets of productions are the original set and resulting set. Dead code (function bigfun as well as the first argument of cons in function f) is highlighted in green.

![Screen dump of the system on a small example of four functions and constructors nil and cons.](image)

**Fig. 7.** A prototype implementation.

**Extensions.** We believe that our method for dead code analysis can be extended to handle side effects. The extension is to use graph grammars instead of tree grammars. The ideas of including L and D as terminals, constructing grammars based on program points as well as the semantics of program constructs connecting these points, and doing grammar simplifications are the same. Recent work by Sagiv, Reps, and Wilhelm [46] uses graph grammars for shape analysis. We believe we can make similar use of graph grammars for dead code analysis in the presence of destructive updates.

Our method can also be extended to handle higher-order functions in two ways, and we have worked out this extension in the second way. First, we can simply apply a control-flow analysis [50] before we do dead code analysis. This
allows our method to handle complete programs that contain higher-order functions. Second, we can directly construct productions corresponding to function abstraction and application and add rules for simplifying them. This is similar to how Henglein [20] addresses higher-order binding-time analysis and how Heintze [19] handles higher-order functions for analyzing sets of values for ML programs. Similar use of constraints has been studied for stopping deforestation for higher-order programs [49]. Our extension adds two constraints/productions for each lambda expression and uses two additional rules for simplification; it is not yet implemented. Handling higher-order functions does not increase the time complexity of our algorithms. In fact, for a language with higher-order functions but not recursive data construction, the constraints may be simplified in worst-case almost linear time [20].

Our method is described here for an untyped language, but the analysis results provide an important kind of type information; the analysis may also be adopted to enhance soft typing and the analysis applies to typed languages as well. For example, consider the third set of productions in Example 2. The grammar at each program point gives its liveness together with the shape of data. Dead code should be reported to the programmer before, or at least at the same time as, type errors such as \texttt{3rd(cons(1, 2))} in the dead code. Live code may have its type inferred by small refinements of our rules. For example, if we replace \texttt{L} by \texttt{Boolean} for the condition in rule (3) of Figure 2, we have \( N_{17} \rightarrow \texttt{Boolean} \) in the third set of productions in Example 2, and thus everything there is precisely typed. Finally, for a typed language, possible values are restricted also by type information, so the overall analysis results can be more precise, e.g., type information about the value of an expression \( e \) can help restrict the grammar at \( e \) when \( e \) is the argument of a primitive function \( e \).

## 7 Related work and conclusion

Our backward dependence analysis uses liveness patterns, which are domain projections, to specify sufficient information. Wadler and Hughes use projections for strictness analysis [56]. Their analysis is also backward but seeks necessary rather than sufficient information, and it uses a fixed finite abstract domain for all programs. Launchbury uses projections for binding-time analysis of partially static data structures in partial evaluation [27]. It is a forward analysis equivalent to strictness analysis and uses a fixed finite abstract domain as well [28]. Mogensen [39], De Niel, and others [11] also use projections, based on grammars in particular, for binding-time analysis and program bifurcation, but they use only a restricted class of regular tree grammars. Another kind of analysis is escape analysis [42, 13, 4], but existing methods can not express as precise information as we do.

Several analyses are in the same spirit as ours. The necessity interpretation by Jones and Le Métayer [24] uses necessity patterns that correspond to a restricted class of liveness patterns. Necessity patterns specify only heads and tails of list values. The absence analysis by Hughes [21] uses contexts that correspond to a restricted class of liveness patterns. Even if it is extended for recursive data types, it handles only a finite domain of list contexts where every head context and every tail context is the same. The analysis for pruning by Liu, Stoller, and Teitelbaum [33] uses projections to specify specific components of tuple
values and thereby provide more accurate information. However, methods used
there for handling unbounded growth of such projections are crude. Wand and
Siveroni's recent work [58] discusses safe elimination of dead variables but does
not handle data constructions. Our method of replacing all dead code (including
dead variables) by a dummy constant — is simple, direct, and more general than
their method; in particular, it is safe to simply remove dead function parameters.

The idea of using regular tree grammars for program flow analysis is due to
Jones and Muchnick [22], where it is used mainly for shape analysis and hence
for improving storage allocation. It is later used to describe other data flow
information such as types and binding times [38, 39, 2, 11, 59, 51, 45]. In particular,
the analysis for backward slicing by Reps and Turnidge [45] explicitly adopts
regular tree grammars to represent projections. It is closest in goal and scope
to our analysis. However, it uses only a limited class of regular tree grammars,
in which each nonterminal appears on the left side of one production, and each
right side is one of five forms, corresponding to \( L, D \), atom, pair, and atom [pair].
It forces grammars to be deterministic in a most approximate way, and it gives
no algorithms for computing the least fixed point from the set of equations. Our
work uses general regular tree grammars extended with \( L \) and \( D \). We also use
productions of extended forms to make the framework more flexible. We give
efficient algorithms for constructing and simplifying the grammars. Compared
with [45], we also handle more program constructs, namely, binding expressions
and user-defined constructors of arbitrary arity.

Our treatment is rigorous, since we have adopted the view that regular-tree-
grammar-based program analysis is also abstract interpretation and approximations
can be built into the grammar transformers as a set of constraints [9]. We
extend the grammars and handle \( L \) and \( D \) specially in grammar manipulations.
The result can also be viewed as using program-based finite grammar domains for
yielding precise and efficient analysis methods. Another standard way to obtain
the analysis result is to do a fixed point computation using general grammar
transformers on potentially infinite grammar domains and use approximation
operations to guarantee termination. Approximation operations provide a more
general solution and make the analysis framework more modular and flexible
[9]. In a separate paper [30], we describe three approximation operations that
together produce significantly more precise analysis results than previous meth-
ods. Each operation is efficient, but due to their generality and interaction, that
work does not have an exact characterization of the total number of iterations
needed. The finite domains described in this work make a complete analysis easy,
and it yields a most precise liveness pattern for the data at each program point.

Regular-tree-grammar-based program analysis can be reformulated as set-
constraint-based analysis [18, 19, 9], but we do not know any work that treats
precise and efficient dead code analysis for recursive data as we do. Melski
and Reps [35, 36] show the interconvertibility of a class of set constraints and
context-free-language reachability and, at the end of [35], they show how gen-
eral CFL-reachingness can be applied nicely to program slicing. That essentially
addresses the same problem we do with a similar framework, but their descrip-
tion is sketchy, with little discussion about correctness and with no results from
implementation, experiments, or applications.

The method and algorithms for dead code elimination studied here have many
applications: program slicing and specialization [60, 45], strength reduction, finite
differencing, and incrementalization [7, 41, 34, 32], caching intermediate results
for program improvement [33], deforestation and fusion [55,6], as well as compile-time garbage collection [24,21,42,57]. The analysis results also provide a kind of type information.

The overall goal of this work is to analyze dead data and eliminate computations of them across recursions and loops, possibly interleaved with wrappers such as classes in object-oriented programs. This paper discusses techniques for recursion. The basic ideas should extend to loops. Pugh and Rosser's work has started this direction; it extends slicing to symbolically capture particular iterations in a loop [43]. Object-oriented programming is used widely, but cross-class optimization heavily depends on inlining, which often causes code blow-up. Grammar-based analysis and transformation can be applied to methods across classes without inlining. A direct application would be to improve techniques for eliminating dead data members, as noted by Sweeney and Tip [53].

Even though this paper focuses on dead code analysis and dead code elimination for recursive data, the framework for representing recursive substructures using general regular tree grammars and the algorithms for computing them applies to other analyses and optimizations on recursive data as well, e.g., binding-time analysis for partial evaluation [27,39]. We have recently developed a binding-time analysis using the same framework.

References