Eliminating Dead Code on Recursive Data *

Yanhong A. Liu, Scott D. Stoller

Computer Science Department, State University of New York at Stony Brook
Stony Brook, NY 11794, USA
E-mail: {liu, stoller}@cs.sunysb.edu

Abstract

This paper describes a powerful method for dead code analysis and elimination in the presence of recursive data constructions. We describe 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 constraints constructed from the program and programming language semantics. The analysis yields the most precise program-based grammars that satisfy the constraints. 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 framework for representing and analyzing properties of recursive data structures using general regular tree grammars applies to other analyses as well.

Key words: Dead-code elimination, Recursive data structures, Regular-tree grammars, Constraints, Program analysis, Slicing

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 [45,1]. There are also other programming activities that do not explicitly involve live or dead code but rely on similar notions. Examples are program slicing [64,50], specialization [50], incrementalization [39,38], and

* This work was supported in part by NSF under grant CCR-9711253 and ONR under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109.

compile-time garbage collection [28,24,47,61]. Analysis for identifying dead code, or code having similar properties, has been studied and used widely [8,7,29,46,1,28,24,10,30,39,58,50,38,61,62]. 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 [24,10,30,50,5,38]. Since recursive data constructions are used increasingly widely in high-level languages [56,14,42,3], an important problem is to identify *partially dead recursive data*—that is, recursive data whose dead parts form recursive substructures—and eliminate code that computes them. [1] 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 [28,24,50,38], but all have limitations.

This paper describes a powerful method for analyzing and eliminating dead code in the presence of recursive data constructions. We describe 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 constraints constructed from the program and programming language semantics. The analysis yields the most precise program-based grammars that satisfy the constraints. 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 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 grammar-based liveness patterns for describing partially dead recursive data. Section 4 formulates the analysis as solving sets of constraints on liveness patterns at all program points. Section 5 presents efficient algorithms for computing the most precise program-based grammars that satisfy the constraints. Section 6 describes dead code elimination, our implementation, and extensions. Section 7 compares this work with related work and concludes.

[1] This is different from *partial dead code*, which is code that is dead on some but not all computation paths [30,5].

2 Language

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

$$
\begin{array}{rll}
e ::= & v & \text{variable} \\
| & c(e_1, ..., e_n) & \text{constructor application} \\
| & c_{\bar{i}}(e_1) & \text{selector application} \\
| & c?(e_1) & \text{tester application} \\
| & p(e_1, ..., e_n) & \text{primitive function application} \\
| & \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 & \text{conditional expression} \\
| & \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 & \text{binding expression} \\
| & f(e_1, ..., e_n) & \text{function application}
\end{array}
$$

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 tester $c?$ that tests whether the argument is an application of c, and if $n > 0$, then for each $i = 1..n$, there is a selector $c_{\bar{i}}$ that selects the ith component in an application of c, e.g., $c_{\bar{2}}(c^3(x, y, z)) = y$. A program is a set of mutually recursive function definitions of the form:

$$f(v_1, ..., v_n) \ \triangleq\ e \tag{1}$$

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

This language has call-by-value semantics. Well-defined expressions evaluate to constructed data, such as $cons(3, nil)$. We use $\perp$ to denote the value of undefined (including non-terminating) expressions; an expression must evaluate to $\perp$ if any of its subexpressions evaluates to $\perp$. Since a program can use data constructions $c(e_1, ..., e_n)$ 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 values of interest to the user,

$$
\begin{array}{lll}
minmax(x) & : & \text{compute min and max for all suffixes of } x \\
minmax(x) & \triangleq & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else if } null(cdr(x)) \textbf{ then} \\
& & \quad cons(triple(car(x), car(x), car(x)), nil) \\
& & \textbf{else let } v = minmax(cdr(x)) \textbf{ in} \\
& & \quad cons(triple(car(x), \\
& & \qquad\qquad \min(car(x), 2nd(car(v))), \\
& & \qquad\qquad \max(car(x), 3rd(car(v)))), \\
& & \qquad v) \\
getsecond(x) & : & \text{get the second component of each triple in } x \\
getsecond(x) & \triangleq & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else } cons(2nd(car(x)), getsecond(cdr(x))) \\
getmin(x) & : & \text{compute the min elements for all suffixes of } x \\
getmin(x) & \triangleq & getsecond(minmax(x))
\end{array}
$$

$$
\begin{array}{lll}
len(x) & : & \text{compute length of } x \\
len(x) & \triangleq & \textbf{if } null(x) \textbf{ then } 0 \\
& & \textbf{else } 1 + len(cdr(x)) \\
odd(x) & : & \text{get elements of } x \text{ at odd positions} \\
even(x) & : & \text{get elements of } x \text{ at even positions} \\
odd(x) & \triangleq & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else } cons(car(x), \\
& & \qquad even(cdr(x))) \\
even(x) & \triangleq & \textbf{if } null(x) \textbf{ then } nil \\
& & \textbf{else } odd(cdr(x))
\end{array}
$$

Fig. 1. Example function definitions.

e.g., in obtaining the return value of a function or certain part of the return value. To improve program efficiency, readability of relevant code, etc., we can eliminate code that produces such dead data and use a special symbol $\_$ as a placeholder for the dead data. A constructor application does not evaluate to $\_$ even if some arguments evaluate to $\_$. A selector, tester, or primitive function application (or a conditional expression) must evaluate to $\_$, if not $\bot$, if any of its arguments (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 the function definition (or the body, respectively).

Values of interest to the user can be either specified by a user using liveness patterns described below or determined by how these results are used in computing other values, e.g., by how the value of *minmax* is used in computing *getmin*, in which case all the max operations are dead.

3 Liveness patterns

We describe partially dead recursive data using liveness patterns. A liveness pattern indicates which parts of data must be dead and which parts may be live. D indicates that a part must be dead, and L indicates that a part may be live. Partial liveness is described using constructors. For example, $cons(D, L)$ indicates a *cons* structure with a definitely dead head and a possibly live tail. Also, the liveness pattern $nil()$ indicates a *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,

$$cons(D, cons(L, D))\,(cons(0, cons(1, cons(2, nil)))) = cons(\_, cons(1, \_)).$$

Formally, liveness patterns are domain projections [52,19], which provide a clean tool for describing substructures of constructed data by projecting out the parts that are of interest [60,31,44,50,38]. Let X be the domain of all possible values computed by our programs, including $\perp$ and values containing $\_$. We define a partial order $\sqsubseteq$ on X, where we read $x_1 \sqsubseteq x_2$ as "x_2 is more live than x_1": for all x in X, $\perp \sqsubseteq x$, and for two values x_1 and x_2 other than $\perp$,

$$\begin{array}{l} x_1 \sqsubseteq x_2 \quad \text{iff} \;\; x_1 = \_, \;\text{ or }\;\; x_1 = x_2, \;\text{ or} \\ \qquad x_1 = c(x_{11}, ..., x_{1n}),\; x_2 = c(x_{21}, ..., x_{2n}),\; \text{and } x_{1i} \sqsubseteq x_{2i} \text{ for } i = 1..n. \end{array} \tag{2}$$

A *liveness pattern* over X is a function $\pi : X \to X$ 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: $D(x) = \_$ for all $x \neq \perp$, and $D(\perp) = \perp$. $c^n(\pi_1, ..., \pi_n)$ is the function:

$$c^n(\pi_1, ..., \pi_n)(x) = \begin{cases} c^n(\pi_1(x_1), ..., \pi_n(x_n)) & \text{if } x = c^n(x_1, ..., x_n) \\ \perp & \text{otherwise} \end{cases} \tag{3}$$

For convenience of presenting the analysis later, we define C to be the function that projects out all constructors but not their arguments:

$$C(x) = \begin{cases} c^n(\_, ..., \_) & \text{if } x = c^n(x_1, ..., x_n) \\ \perp & \text{otherwise} \end{cases} \tag{4}$$

We define a partial order $\leq$ on liveness patterns, where we read $\pi_1 \leq \pi_2$ as "π_2 projects out more live data than π_1 does":

$$\pi_1 \leq \pi_2 \quad \text{iff} \quad \forall x,\; \pi_1(x) \sqsubseteq \pi_2(x) \tag{5}$$

For convenience, we define $\pi_1 \geq \pi_2$ if and only if $\pi_2 \leq \pi_1$.

Grammar-based liveness patterns. Since a program can produce recursive data of unbounded size, we represent liveness patterns as grammars. For example, the grammar $S \to nil \,|\, cons(D, S)$ projects out a list whose elements are dead but whose spine is live. It generates the set of sentences

$\{nil(), cons(D, nil()), cons(D, cons(D, nil())), ...\}$. Applying each element to a given value, say, $cons(2, cons(4, nil))$, yields $\{\bot, \bot, cons(\_, cons(\_, nil)), \bot, ...\}$, of which $cons(\_, cons(\_, nil))$ is the least upper bound.

Formally, the grammars we use to represent liveness patterns are regular tree grammars [18], which allow bounded, and often precise, representations of unbounded structures [27,43,44,2,55,9,50]. A *regular-tree-grammar-based liveness pattern*, called *liveness grammar* in short, denoted G, is a quadruple $\langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S\rangle$, where $\mathcal{T}$ is a set of terminal symbols including L, D, and all possible constructors c, $\mathcal{N}$ is a set of nonterminal symbols N, $\mathcal{P}$ is a set of productions of the form:

$$N \rightarrow D, \quad N \rightarrow L, \quad \text{or} \quad N \rightarrow c^n(N_1, ..., N_n), \tag{6}$$

and nonterminal S is the start symbol. So, our liveness grammars use general regular tree grammars [27,18,9] extended with the special constants D and L. A sentence generated by a liveness grammar is a liveness pattern. Let language $\mathcal{L}_G$ be the set of sentences generated by G. The projection function that G represents, denoted $[\![G]\!]$, is:

$$[\![G]\!](x) = \sqcup\{\pi(x) \mid \pi \in \mathcal{L}_G\} \tag{7}$$

where $\sqcup$ is the least upper bound operation for $\sqsubseteq$. It is easy to see that $[\![G]\!](x)$ is well-defined for all $x \in X$: each x in X is finite, so $\{\pi(x) \mid \pi \in \mathcal{L}_G\}$ is finite, and thus the least upper bound exists. Note that a grammar that generates sentence L represents the projection function L, and a grammar that generates only sentence D represents the projection function D. For ease of presentation, when no confusion arises, we write grammars in compact forms. For example, $\{S \rightarrow nil \mid cons(N, S),\ N \rightarrow triple(D, L, D)\}$, where $\mid$ denotes alternation, projects out a list whose elements are triples whose first and third components are dead.

We extend liveness grammars to allow productions of the form:

$$N \rightarrow N', \quad N \rightarrow c_i^-(N'), \quad \text{or} \quad N \rightarrow [N']\,R' \tag{8}$$

for R' of the form L, $c^n(N_1, ..., N_n)$, or N'', and we define the corresponding functions:

$$c_i^-(\pi) = \begin{cases} L \text{ if } \pi = L \\ \pi_i \text{ if } \pi = c^n(\pi_1, ..., \pi_n) \\ D \text{ otherwise} \end{cases} \qquad [\pi]\pi' = \begin{cases} D \text{ if } \pi = D \\ \pi' \text{ otherwise} \end{cases} \tag{9}$$

These extended forms are introduced for convenience of presenting the analysis later; the selector form in the middle of (8) is the same as that first used by Jones and Muchnick [27], and the conditional form on the right of (8) is for similar purposes as those used in several other analyses, e.g., the operator $\triangleright$ used by Wadler and Hughes for strictness analysis [60]. Given an extended liveness grammar G that contains productions of the forms in (6) and (8), we can construct a liveness grammar G' that contains only productions of the forms in (6) such that $[\![G]\!] = [\![G']\!]$, 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 given by a user to specify values of interest or are constructed based on the program and programming language semantics, and then simplify the grammars to equivalent grammars in simpler forms where, in particular, the only grammar that represents the projection function D is $\{S \to D\}$.

4 Analysis of liveness patterns using constraints

Values of interest to the user can be specified as liveness patterns associated with *program points*, such as function definitions, parameters, and subexpressions. For example, for the functions in Figure 1, a user may specify a liveness grammar $\{S \to L\}$ associated with the body of function *len*, indicating that the return value of *len* is of interest, or, a user may specify a liveness grammar $\{S \to nil() \mid cons(D, D)\}$ associated with the body of function *odd*, indicating that whether the return value of *odd* is an empty list is of interest. We assume that the given liveness grammars do not contain extended forms.

Given liveness patterns associated with program points of interest, dead code analysis computes liveness patterns at all program points, so that the liveness specified by the given liveness patterns is guaranteed. It is a backward dependence analysis and is aimed at finding minimum liveness patterns. The analysis is based on the idea 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, (Condition 1) the resulting liveness pattern at a program point must project out values that are more live than required by the given liveness pattern, if any, associated with that point. Precisely, at each subexpression e where a liveness pattern π_0 is given, if the resulting liveness pattern at e is π, then $\pi_0(e) \sqsubseteq \pi(e)$ for all values of the free variables in e.

Second, (Condition 2) the resulting liveness patterns must satisfy the constraints determined by the programming language semantics. Precisely, assume a resulting liveness pattern is associated with each parameter and each subexpression of all function definitions. Let ${}^{\pi:}e$ denote that liveness pattern π is associated with e. Then (Condition 2a) 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 $f({}^{\pi_1:}v_1, ..., {}^{\pi_n:}v_n) \triangleq {}^{\pi:}e$, the following sufficiency condition must be satisfied for all values $v_1, ..., v_n$:

$$\pi(f(v_1, ..., v_n)) \sqsubseteq f(\pi_1(v_1), ..., \pi_n(v_n)) \tag{10}$$

and (Condition 2b) 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 of Figure 2, the corresponding sufficiency condition in the right column must be satisfied for all values of the free variables in the subexpression.

${}^{\pi:}c({}^{\pi_1:}e_1, ..., {}^{\pi_n:}e_n)$	$\pi(c(e_1, ..., e_n)) \sqsubseteq c(\pi_1(e_1), ..., \pi_n(e_n))$
${}^{\pi:}c_{\bar{i}}({}^{\pi_1:}e_1)$	$\pi(c_{\bar{i}}(e_1)) \sqsubseteq c_{\bar{i}}(\pi_1(e_1))$
${}^{\pi:}c?({}^{\pi_1:}e_1)$	$\pi(c?(e_1)) \sqsubseteq c?(\pi_1(e_1))$
${}^{\pi:}p({}^{\pi_1:}e_1, ..., {}^{\pi_n:}e_n)$	$\pi(p(e_1, ..., e_n)) \sqsubseteq p(\pi_1(e_1), ..., \pi_n(e_n))$
${}^{\pi:}\textbf{if } {}^{\pi_1:}e_1 \textbf{ then } {}^{\pi_2:}e_2 \textbf{ else } {}^{\pi_3:}e_3$	$\pi(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \sqsubseteq \textbf{if } \pi_1(e_1) \textbf{ then } \pi_2(e_2) \textbf{ else } \pi_3(e_3)$
${}^{\pi:}\textbf{let } u = {}^{\pi_1:}e_1 \textbf{ in } {}^{\pi_2:}e_2$	$\pi(\textbf{let } u = e_1 \textbf{ in } e_2) \sqsubseteq \textbf{let } u = \pi_1(e_1) \textbf{ in } \pi_2(e_2)$
${}^{\pi:}f({}^{\pi_1:}e_1, ..., {}^{\pi_n:}e_n)$	$\pi(f(e_1, ..., e_n)) \sqsubseteq f(\pi_1(e_1), ..., \pi_n(e_n))$

Fig. 2. Sufficiency conditions for expressions.

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:

$$f(x, y) \triangleq \textbf{if } x > 0 \textbf{ then } x \textbf{ else } y$$
$$g(z) \triangleq {}^{\pi_0:}\ f({}^{L:}1, {}^{D:}z) + f({}^{L:}0, {}^{L:}z)$$

Given $\pi_0 = L$ 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.

Constraints. Given liveness patterns associated with program points of interest, we construct a set of constraints on the resulting liveness patterns at all program points that guarantee the sufficiency conditions.

First, at each subexpression e where a liveness pattern is given, if the given liveness pattern is π_0, and the resulting liveness pattern is π, then we construct $\pi \geq \pi_0$.

Second, for a function definition of the form $f({}^{\pi_1}{:}v_1, ..., {}^{\pi_n}{:}v_n) \triangleq e$, we construct, for $i = 1..n$ and for each occurrence ${}^{\pi'_i}{:}v_i$ in e, the constraint

$$\pi_i \geq \pi'_i \tag{11}$$

and, for each subexpression of e that is of a form in the left column of Figure 3, we construct the corresponding constraints in the right column. These constraints make approximations while guaranteeing the sufficiency conditions, as explained below.

$(R1)$	${}^{\pi}{:}c({}^{\pi_1}{:}e_1, ..., {}^{\pi_n}{:}e_n)$	$\pi_i \geq c_i^-(\pi)$ for $i = 1..n$
$(R2)$	${}^{\pi}{:}c_i^-({}^{\pi_1}{:}e_1)$	$\pi_1 \geq [\pi]\, c(D, ..., D, \pi, D, ..., D)$ with $i-1$ D's before π and $n-i$ D's after π
$(R3)$	${}^{\pi}{:}c?({}^{\pi_1}{:}e_1)$	$\pi_1 \geq [\pi]\, C$
$(R4)$	${}^{\pi}{:}p({}^{\pi_1}{:}e_1, ..., {}^{\pi_n}{:}e_n)$	$\pi_i \geq [\pi]\, L$ for $i = 1..n$
$(R5)$	${}^{\pi}{:}\textbf{if } {}^{\pi_1}{:}e_1 \textbf{ then } {}^{\pi_2}{:}e_2 \textbf{ else } {}^{\pi_3}{:}e_3$	$\pi_1 \geq [\pi]\, L,\ \pi_2 \geq \pi,\ \pi_3 \geq \pi$
$(R6)$	${}^{\pi}{:}\textbf{let } u = {}^{\pi_1}{:}e_1 \textbf{ in } {}^{\pi_2}{:}e_2$	$\pi_1 \geq \pi'_1$ for each free occurrence of ${}^{\pi'_1}{:}u$ in e_2, $\pi_2 \geq \pi$
$(R7)$	${}^{\pi}{:}f({}^{\pi_1}{:}e_1, ..., {}^{\pi_n}{:}e_n)$ where $f({}^{\pi'_1}{:}v_1, ..., {}^{\pi'_n}{:}v_n) = {}^{\pi'}{:}e'$	$\pi_i \geq [\pi]\, \pi'_i$ for $i = 1..n$, $\pi' \geq \pi$

Fig. 3. Constraints for expressions.

Formula (11) 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 (R7) 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. In effect, if a function call is dead, then all its arguments are also dead, but the formal parameters of the function might not be dead due to other calls to the same function.

Other constraints are based on the semantics of each construct locally. Rules (R1)-(R3) handle data constructions. Rule (R1) 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 (R1), for any constructor of arity 0, no constraint is added. Rule (R2) requires that, if the result of a selection by c_i is not dead, then the argument be as live as a construction using c whose ith component is as live as the result of the selection. Rule (R3) 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 (R4) 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 C in place of L in the constraints. Rule (R5) 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 C in place of L as a sufficient context for e_1; furthermore, if e_2 equals e_3, in fact as long as $\pi(e_2)$ equals $\pi(e_3)$, then we could use D in place of L as a sufficient context for e_1 and thus no constraint for π_1 would be needed. Rule (R6) 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.

A standard inductive argument can be used to show that solutions to these constraints satisfy the sufficiency conditions. We are interested in solutions expressible using a finite set of liveness patterns determined by the given program, as detailed in Section 5. The set is finite, therefore, with the ordering (5), we have a complete partial order. The constructed system of constraints is equivalent to a system of equations: a constraint $\pi_1 \geq \pi_2$ is rewritten as $\pi_1 = \pi_1 \vee \pi_2$, where $\vee$ is the least upper bound operator for $\leq$. The right sides of these equations are monotonic, so a unique least solution exists [19]. Taking the least solution allows the most dead code to be eliminated.

5 Construction and simplification of liveness grammars

We describe a straightforward method for building the most precise program-based liveness grammars that satisfy the above constraints; these grammars may contain productions of the extended forms in (8). Then, we simplify the grammars by eliminating extended forms; this makes explicit whether the grammar associated with a program point equals dead.

Constructing the grammars. Let $\mathcal{T}_c$ be the set of all possible constructors in the program. Let $\mathcal{N}_0$ be the set of nonterminals used in the given liveness grammars associated with selected subexpressions. We associate a unique nonterminal, not in $\mathcal{N}_0$, 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.

Let ${}^{N}{:}e$ denote that nonterminal N is associated with e. We construct two kinds of productions. First, 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 construct $N \to N_0$ as well as all productions in G_0. Second, for each function definition $f({}^{N_1}{:}v_1, ..., {}^{N_n}{:}v_n) \triangleq e$, we construct, for each $i = 1..n$ and for each occurrence ${}^{N'_i}{:}v_i$ in e, the production

$$N_i \to N'_i \tag{12}$$

and, for each subexpression of e that is of a form in the left column of Figure 4, the corresponding productions in the right column.

${}^{N}{:}c({}^{N_1}{:}e_1, ..., {}^{N_n}{:}e_n)$	$N_i \to c_{\bar{i}}(N)$ for $i = 1..n$
${}^{N}{:}c_{\bar{i}}({}^{N_1}{:}e_1)$	$N_1 \to [N]\, c^n(D, ..., D, N, D, ..., D)$
	with $i-1$ D's before N and $n-i$ D's after N
${}^{N}{:}c?({}^{N_1}{:}e_1)$	$N_1 \to [N]\, c^n(D, ..., D)$
	with n D's for each possible constructor c^n
${}^{N}{:}p({}^{N_1}{:}e_1, ..., {}^{N_n}{:}e_n)$	$N_i \to [N]\, L$ for $i = 1..n$
${}^{N}{:}\textbf{if } {}^{N_1}{:}e_1 \textbf{ then } {}^{N_2}{:}e_2 \textbf{ else } {}^{N_3}{:}e_3$	$N_1 \to [N]\, L,\ N_2 \to N,\ N_3 \to N$
${}^{N}{:}\textbf{let } u = {}^{N_1}{:}e_1 \textbf{ in } {}^{N_2}{:}e_2$	$N_1 \to N'_1$ for each free occurrence of ${}^{N'_1}{:}u$ in e_2, $N_2 \to N$
${}^{N}{:}f({}^{N_1}{:}e_1, ..., {}^{N_n}{:}e_n)$ where $f({}^{N'_1}{:}v_1, ..., {}^{N'_n}{:}v_n) = {}^{N'}{:}e$	
	$N_i \to [N]\, N'_i$ for $i = 1..n$, $N' \to N$

Fig. 4. Productions constructed for expressions.

It is easy to show that the projection functions represented by the resulting grammars satisfy the constraints in Section 4 and thus give sufficient information at every program point. To show this, simply notice that the constructed productions can be obtained from the constraints in Section 4 by replacing π with N, $\geq$ with $\to$, and C with $c(D, ..., D)$ for all c. For grammars G and G' with start symbol N and N', respectively, a production $N \to N'$ ensures that $\mathcal{L}_G \supseteq \mathcal{L}_{G'}$, which implies $[\![G]\!] \geq [\![G']\!]$. Thus, each production constructed here guarantees exactly a corresponding constraint in Section 4 simply by definitions.

The projection functions represented by the resulting grammars are the unique least solution to the constraints in Section 4 among solutions expressible with liveness grammars that use the program-based finite set of nonterminals described above. The set of such liveness grammars is finite, so a unique least solution exists, by the arguments at the end of Section 4. To see that the resulting grammars are the least solution, 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 constraints.

Let n denote the size of the program. Assume that the number of possible constructors at the argument of each tester application is bounded by a constant. For example, for Scheme programs, this number is 2, one for *nil* and one for *cons*; for a typed language like ML, this is the number of alternatives in a data type declaration. Since a constant number of productions are added at each program point, the above construction takes $O(n)$ time.

Example 5.1 For functions *len*, *odd*, and *even* in Figure 1, the nonterminals labeling the program points and the added productions are shown in Figure 5. For example, we have $N_{12} \to [N_{13}]cons(N_{13}, N_0)$, where $N_0 \to D$ is the last production on the last line. It means that N_{12} is conditioned on N_{13}: if N_{13} derives only D, so does N_{12}, otherwise N_{12} derives $cons(N_{13}, N_0)$, i.e., it projects out a *cons* structure, the first component of which is projected out by N_{13}.

$$
\begin{array}{lcl}
len({}^{N_{29}}\!:x) & \triangleq & {}^{N_{28}}\!:\textbf{if } {}^{N_{27}}\!:null({}^{N_{26}}\!:x) \textbf{ then } {}^{N_{25}}\!:0 \\
 & & \quad \textbf{else } {}^{N_{24}}\!:{}^{N_{23}}\!:1 + {}^{N_{22}}\!:len({}^{N_{21}}\!:cdr({}^{N_{20}}\!:x)) \\
odd({}^{N_{19}}\!:x) & \triangleq & {}^{N_{18}}\!:\textbf{if } {}^{N_{17}}\!:null({}^{N_{16}}\!:x) \textbf{ then } {}^{N_{15}}\!:nil \\
 & & \quad \textbf{else } {}^{N_{14}}\!:cons({}^{N_{13}}\!:car({}^{N_{12}}\!:x), {}^{N_{11}}\!:even({}^{N_{10}}\!:cdr({}^{N_{9}}\!:x))) \\
even({}^{N_{8}}\!:x) & \triangleq & {}^{N_{7}}\!:\textbf{if } {}^{N_{6}}\!:null({}^{N_{5}}\!:x) \textbf{ then } {}^{N_{4}}\!:nil \\
 & & \quad \textbf{else } {}^{N_{3}}\!:odd({}^{N_{2}}\!:cdr({}^{N_{1}}\!:x))
\end{array}
$$

$N_{29} \to N_{26},\ N_{29} \to N_{20},\ N_{26} \to [N_{27}]\,cons(N_0, N_0), N_{26} \to [N_{27}]nil,\ N_{27} \to [N_{28}]L,\ N_{25} \to N_{28},$
$N_{23} \to [N_{24}]L,\ N_{20} \to [N_{21}]\,cons(N_0, N_{21}),\ N_{21} \to [N_{22}]N_{29},\ N_{28} \to N_{22},\ N_{22} \to [N_{24}]L,\ N_{24} \to N_{28},$

$N_{19} \to N_{16},\ N_{19} \to N_{12},\ N_{19} \to N_9,\ N_{16} \to [N_{17}]\,cons(N_0, N_0),\ N_{16} \to [N_{17}]nil,\ N_{17} \to [N_{18}]L,$
$N_{15} \to N_{18},\ N_{12} \to [N_{13}]cons(N_{13}, N_0),\ N_{13} \to car(N_{14}),\ N_9 \to [N_{10}]\,cons(N_0, N_{10}),\ N_{10} \to [N_{11}]N_8,$
$N_7 \to N_{11},\ N_{11} \to cdr(N_{14}), N_{14} \to N_{18},$

$N_8 \to N_5,\ N_8 \to N_1,\ N_5 \to [N_6]\,cons(N_0, N_0),\ N_5 \to [N_6]\,nil,\ N_6 \to [N_7]\,L,\ N_4 \to N_7,$
$N_1 \to [N_2]\,cons(N_0, N_2),\ N_2 \to [N_3]\,N_{19}, N_{18} \to N_3,\ N_3 \to N_7,\ N_0 \to D$

Fig. 5. Productions constructed for the example functions.

Suppose we need the result of *len*; we add $N_{28} \to L$, since N_{28} corresponds to the return value of *len*. Suppose we need the result of *odd*; we add $N_{18} \to L$. Suppose we need to know whether the result of *odd* is *nil* or *cons*; we add $N_{18} \to nil$ and $N_{18} \to cons(D, D)$.

Simplifying the grammars. The grammars obtained above may contain productions of the extended forms in (8) and thus be difficult to understand and use. We simplify the grammars by removing extended forms using the iterative algorithm in Figure 6. After the simplification, nonterminals that

input: a grammar $\langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$
/* assume R is of the form L or $c(N_1, ..., N_n)$, and R' is of the form L, $c(N_1, ..., N_n)$, or N'' */
repeat
  if $\mathcal{P}$ contains $N \to N'$ and $N' \to R$, then add $N \to R$ to $\mathcal{P}$;
  if $\mathcal{P}$ contains $N \to c_i^-(N')$ and $N' \to L$, then add $N \to L$ to $\mathcal{P}$;
  if $\mathcal{P}$ contains $N \to c_i^-(N')$ and $N' \to c^n(N_1, ..., N_n)$, then add $N \to N_i$ to $\mathcal{P}$;
  if $\mathcal{P}$ contains $N \to [N']R'$ and $N' \to R$, then add $N \to R'$ to $\mathcal{P}$;
until no more productions can be added;
remove all productions of the extended forms from $\mathcal{P}$;
return simplified grammar $\langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$

Fig. 6. Algorithm for simplifying the grammars.

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 then read off the grammar at any function parameter or subexpression by starting with the associated nonterminal and collecting all productions whose left sides are reachable from this start symbol.

The correctness of the algorithm can be proved in a similar way to when only the selector form is used [27]. The idea is to show, based directly on the definitions of the extended forms, that the original and simplified grammars represent the same projection function. In particular, every nonterminal generates a non-empty language, since it derives at least D, if nothing else; therefore, in the third rule for adding productions, it is not necessary to test whether nonterminals other than N_i generate non-empty languages.

Nonterminals are associated with program points, so there are $O(n)$ of them. Each step adds a production of the form $N \to L$, $N \to c(N_1, ..., N_k)$, or $N \to N'$. Since each right side of the form $c(N_1, ..., N_k)$ 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 and supported by our recent work on the formal design and analysis of the simplification algorithm [35].

Example 5.2 Suppose we need the result of *len* and therefore added $N_{28} \to L$; we obtain the productions in Figure 7(a). Suppose we need the result of *odd* and therefore added $N_{18} \to L$; we obtain the productions in Figure 7(b).

Suppose we added $N_{18} \to nil,\ N_{18} \to cons(D, D)$; we obtain the productions in Figure 7(c). In each case, other nonterminals derive only D.

$N_{29} \to nil,\ N_{29} \to cons(N_0, N_0),\ N_{29} \to cons(N_0, N_{21}),\ N_{28} \to L,\ N_{27} \to L,\ N_{26} \to nil,$
$N_{26} \to cons(N_0, N_0),\ N_{25} \to L,\ N_{24} \to L,\ N_{23} \to L,\ N_{22} \to L, N_{21} \to nil,\ N_{21} \to cons(N_0, N_0),$
$N_{21} \to cons(N_0, N_{21}),\ N_{20} \to cons(N_0, N_{21})$

(a)

$N_{19} \to cons(N_0, N_0),\ N_{19} \to nil,\ N_{19} \to cons(N_{13}, N_0),\ N_{19} \to cons(N_0, N_{10}),\ N_{18} \to L,$
$N_{17} \to L,\ N_{16} \to nil,\ N_{16} \to cons(N_0, N_0),\ N_{15} \to L,\ N_{14} \to L,\ N_{13} \to L,\ N_{12} \to cons(N_{13}, N_0),$
$N_{11} \to L,\ N_{10} \to nil,\ N_{10} \to cons(N_0, N_0),\ N_{10} \to cons(N_0, N_2),\ N_9 \to cons(N_0, N_{10}),$
$N_8 \to cons(N_0, N_0),\ N_8 \to nil,\ N_8 \to cons(N_0, N_2),\ N_7 \to L,\ N_6 \to L,\ N_5 \to nil,$
$N_5 \to cons(N_0, N_0),\ N_4 \to L,\ N_3 \to L,\ N_2 \to cons(N_0, N_{10}),\ N_2 \to cons(N_{13}, N_0),\ N_2 \to nil,$
$N_2 \to cons(N_0, N_0),\ N_1 \to cons(N_0, N_2)$

(b)

$N_{19} \to cons(N_0, N_0),\ N_{19} \to nil,\ N_{18} \to nil,\ N_{18} \to cons(N_0, N_0),\ N_{17} \to L,\ N_{16} \to nil,$
$N_{16} \to cons(N_0, N_0),\ N_{15} \to cons(N_0, N_0),\ N_{15} \to nil,\ N_{14} \to cons(N_0, N_0),\ N_{14} \to nil$

(c)

Fig. 7. Simplified grammars for the example functions.

The resulting grammars can be further simplified by minimization [18], 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 nonterminals derive only D, i.e., whose associated liveness patterns are D. These parts of the program are dead, so we eliminate them by replacing them with $\_$. If the entire body of a function is dead, then we eliminate the function definition.

Example 6.1 Suppose we need to know whether the result of *odd* is *nil* or *cons* and therefore added $N_{18} \to nil,\ N_{18} \to cons(D, D)$; eliminating dead code based on the simplified grammar in Example 5.2, where N_1 to N_{13} all have only D on the right hand sides, yields the function in Figure 8(a). Suppose we need the result of function *getmin*, given in Figure 1; analyzing and eliminating dead code yields the function in Figure 8(b) along with functions *getsecond* and *getmin*. As another example, called *getlen*, if the result of *minmax* is used as argument to *len* instead of *getsecond*, then our algorithm finds that the entire *triple* constructions in *minmax* are dead. However, if the result of *minmax* is used as argument to *odd*, then none of the subexpressions in *minmax* is dead, since the triple is used in every odd recursive call.

$$odd(x) \triangleq \textbf{if } null(x) \textbf{ then } nil \textbf{ else } cons(\_,\_)$$

(a)

$$\begin{aligned} minmax(x) \triangleq\ & \textbf{if } null(x) \textbf{ then } nil \\ & \textbf{else if } null(cdr(x)) \textbf{ then } cons(triple(\_, car(x), \_), nil) \\ & \textbf{else let } v = minmax(cdr(x)) \textbf{ in } cons(triple(\_, \min(car(x), 2nd(car(v))), \_), v) \end{aligned}$$

(b)

Fig. 8. Resulting example functions after dead code elimination.

Our dead code elimination preserves semantics in the sense that, if the original program terminates with a value, then the resulting program terminates a projection of the value that is more live than the required projection of the value, as stated below. In particular, if the required projection is the entire return value, then the new program terminates with the same value.

Theorem 6.1 *Let G_0 be a liveness grammar associated by the user with the body (and thus the return value) of some function f in a given program. Let f' be the corresponding function in the resulting program (if it exists, otherwise G_0 must derive only D). Let $f(e_1, \ldots, e_n)$ be a closed expression. If evaluation of $f(e_1, \ldots, e_n)$ terminates with a value v, then evaluation of $f'(e_1, \ldots, e_n)$ terminates with a value v' such that $[\![G_0]\!](v) \sqsubseteq v' \sqsubseteq v$.*

Proof sketch: Evaluation of an expression in an environment can be represented as a derivation tree built from instances of rules of a straightforward structural operational semantics [65] of our language. Associated with each node is a configuration $\langle e, \rho \rangle$, where e is the expression being evaluated, and ρ is the environment in which e is being evaluated. Let $\rho(e)$ denote the result of evaluating e in ρ.

Using the sufficiency conditions in Section 4, one can show by structure induction on derivation trees that (1) the derivation tree T' for $f'(e_1, \ldots, e_n)$ is isomorphic to the subtree of the derivation tree T for $f(e_1, \ldots, e_n)$ obtained by eliminating subtrees corresponding to evaluation of subexpressions whose associated liveness grammar derives only D, and (2) the configuration $\langle e', \rho' \rangle$ associated with a node in T' is related to the configuration $\langle e, \rho \rangle$ associated with the corresponding node in T by $[\![G]\!](\rho(e)) \sqsubseteq \rho'(e') \sqsubseteq \rho(e)$, where G is the grammar associated with e by the analysis. QED

Two further optimizations are possible but need further study. First, *minmax* in Figure 8(b) can be further optimized by removing the *triple* constructions and selectors. Second, when the result of *minmax* is used as argument to *odd*, there is no dead code in *minmax*, but the triple in every even call is indeed dead. One needs to unfold the definition of *minmax* to remove such *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.

Implementation. We have implemented the analysis in a prototype system. The implementation uses the Synthesizer Generator [49]. 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 [38]. 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 [39]. 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 9 summarizes the experimental results for a number of examples. Program `minmax` is as in Figure 1. Programs `incsort` and `incout` are incremental programs for selection sort and outer product, respectively, derived using incrementalization [39], where dead code after incrementalization is to be eliminated. Programs `cachebin` and `cachelcs` are dynamic-programming programs for binomial coefficients and longest common subsequences, respectively, derived using cache-and-prune [38,36], where cached intermediate results that are not used are 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 [25].

When using dead code analysis for incrementalization and for pruning unused

program name	function of interest	total program points	dead program points	live program points	number of initial productions	number of resulting productions	analysis time (ms) (incl. GC)	analysis time (ms) (excl. GC)
`minmax`	`getlen`	81	50	31	111	90	0.007	0.005
`minmax`	`getmin`	81	32	49	111	150	0.016	0.011
`incsort`	`sort'`	108	84	24	143	34	0.006	0.005
`incout`	`out'`	117	62	55	151	78	0.007	0.006
`cachebin`	`bin`	74	7	67	90	114	0.014	0.010
`cachelcs`	`lcs`	101	12	89	139	206	0.038	0.033
`calend`	`gregorian2absolute`	1551	1359	192	1839	229	0.067	0.053
`calend`	`islamic-date`	1551	1205	346	1839	419	0.083	0.069
`calend`	`eastern-orthodox-Xmas`	1551	1176	375	1839	461	0.086	0.069
`calend`	`yahrzeit`	1551	1123	428	1839	485	0.086	0.068
`takr`	`run-takr`	2804	0	2804	4005	2805	0.403	0.304
`takr`	`tak99`	2804	4	2800	4005	2801	0.419	0.310

Fig. 9. Experimental results.

intermediate results, there is always a particular function of interest, shown in Figure 9. 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 `calend`, much dead code is found, whereas for `takr`, all 100 functions other than the driver function `run-takr`, are involved in calling each other. 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 a Sun station Ultra 10 with 299 MHz 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 intuitive 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. Similar ideas have been studied previously for constraint simplification, e.g., [20,15]. We have recently studied the precise specification, design, analysis, implementation, and measurements of our simplification algorithm and showed that the simplification time is linear in the number of live program points and in other parameters that are typically small in practice [35].

Figure 10 is a screen dump of the system on a small example. Program points are annotated with nonterminals (`Ni`'s) 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.

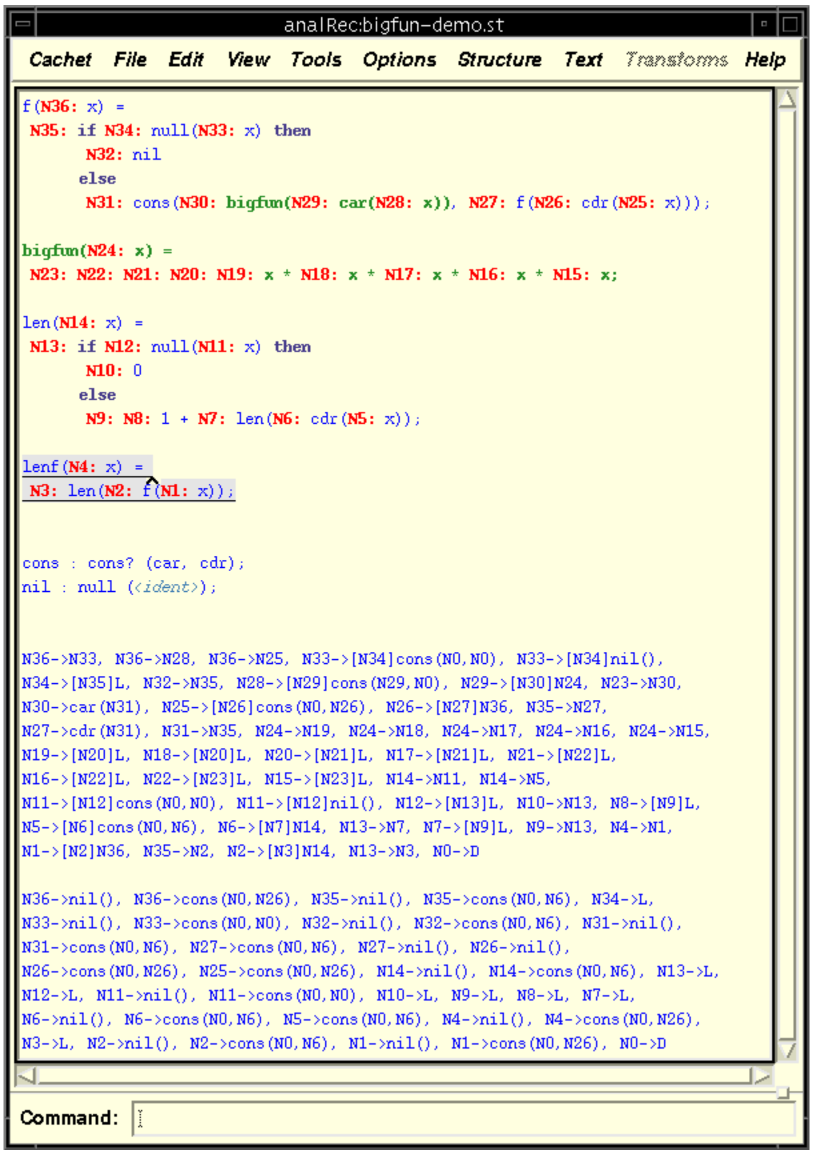

Fig. 10. A prototype implementation.

Extensions. We believe that our method for dead code analysis can be extended to handle side effects. The extension may use graph grammars instead of tree grammars. The ideas of including L and D as terminals, constructing program-based grammars, and doing grammar simplifications should be similar. Sagiv, Reps, and Wilhelm [51] handle destructive updates for shape analysis using shape graphs like a kind of graph grammars. We may 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. First, we can simply apply a control-flow analysis [54] 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 like how Henglein [23] addresses higher-order binding-time analysis and how Heintze [22] and Flanagan and Felleisen [16] handle higher-order functions for analyzing ML programs and Scheme programs, respectively. Use of constraints has also been studied for stopping deforestation for higher-order programs [53].

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 productions in Figure 7(c). 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 $3rd(cons(1,2))$ in the dead code. Live code may have its type inferred by small refinements of our rules. For example, if we replace L with $Boolean$ in the rules for conditional expressions in Figures 3 and 4, where $Boolean$ projects out only true and false, then we have $N_{17} \rightarrow Boolean$ instead of $N_{17} \rightarrow L$ in Figure 7(c), and thus everything there is precisely typed. 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 tester $c?$.

7 Related work and conclusion

Our backward dependence analysis specifies sufficient information using liveness patterns, which are domain projections, that are based on general regular tree grammars and are determined by the given program. Wadler and Hughes use projections for strictness analysis [60]. Their analysis is also backward; it seeks necessary rather than sufficient information and uses a fixed finite abstract domain for all programs. Launchbury uses projections for binding-time analysis of partially static data structures in partial evaluation [31]. It is a forward analysis equivalent to strictness analysis and uses a fixed finite abstract domain as well [32]. Mogensen [44], 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 [47,13,4], but the methods there use data access paths, such as list spines, that have to be cut at certain level for approximation and thus are generally less precise than methods that use grammars.

Several analyses are in the same spirit as ours, but they do not use grammars and are thus less powerful. The necessity interpretation by Jones and Le Métayer [28] 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 [24] 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 [38] 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 [62] discuss safe elimination of dead variables but does not handle data constructions. Our method of replacing all dead code (including dead variables) by a 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 [26], where it is used mainly for shape analysis and then for improving storage allocation. It is later used to describe other data flow information such as types and binding times [43,44,2,11,63,55,50]. In particular, the analysis for backward slicing by Reps and Turnidge [50] 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, which together yields more precise analysis results. Compared with [50], we also handle more program constructs, namely, binding expressions and user-defined constructors of arbitrary arity.

In our treatment, 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 obtaining 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 [34], we describe three approximation operations that together produce significantly more precise analysis results than previous methods. 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 complexity analysis easy.

Regular-tree-grammar-based program analysis can be reformulated as set-constraint-based analysis [21,22,9], but we do not know any work that treats precise and efficient dead code analysis for recursive data as we do. Melski and Reps [40,41] show the interconvertibility of a class of set constraints and context-free-language reachability and, at the end of [41], they show how general CFL-reachability can be applied to program slicing. That essentially addresses the same problem we do with a similar framework, but their description 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 [64,50], strength reduction, finite differencing, and incrementalization [7,46,39,37], caching intermediate results for program improvement [38], deforestation and fusion [59,6], as well as compile-time garbage collection [28,24,47,61]. 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 [48]. 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 [57].

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 [31,44]. We have recently developed a binding-time analysis using the same framework.

Acknowledgements

We would like to thank the anonymous referees for both this issue and SAS'99, whose comments and suggestions helped improve many aspects of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[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] K. Arnold and J. Golsing. *The Java Programming Language*. Addison-Wesley, Reading, Mass., 1996.

[4] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 25–37. ACM, New York, Jan. 1998.

[5] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 159–170. ACM, New York, June 1997.

[6] W.-N. Chin. Safe fusion of functional expressions. In LFP 1992 [33], pages 11–20.

[7] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Commun. ACM*, 20(11):850–856, Nov. 1977.

[8] J. Cocke and J. T. Schwartz. Programming languages and their compilers; preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[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] R. Cytron, J. Ferrante, B. K. Rosen, M. M. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[11] A. De Niel, E. Bevers, and K. De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 142–153. ACM, New York, June 1991.

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

[13] A. Deutsch. On the complexity of escape analysis. In *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 358–371. ACM, New York, Jan. 1997.

[14] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[15] 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.

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

[17] *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, Sept. 1989.

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

[19] C. A. Gunter. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., 1992.

[20] 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.

[21] N. Heintze. *Set-Based Program Analysis*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Oct. 1992.

[22] 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.

[23] 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.

[24] J. Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, pages 117–153. Addison-Wesley, Reading, Mass., 1990.

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

[26] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 244–256. ACM, New York, Jan. 1979.

[27] 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.

[28] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [17], pages 54–74.

[29] K. Kennedy. Use-definition chains with applications. *J. Comput. Lang.*, 3(3):163–179, 1978.

[30] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158. ACM, New York, June 1994.

[31] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, Glasgow, Scotland, 1989.

[32] J. Launchbury. Strictness and binding-time analysis: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 80–91. ACM, New York, June 1991.

[33] *Proceedings of the 1992 ACM Conference on LISP and Functional Programming.* ACM, New York, June 1992.

[34] Y. A. Liu. Dependence analysis for recursive data. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 206–215. IEEE CS Press, Los Alamitos, Calif., May 1998.

[35] Y. A. Liu, N. Li, and S. D. Stoller. Solving regular tree grammar based constraints. In *Proceedings of the 8th International Static Analysis Symposium.* Springer-Verlag, Berlin, July 2001. To appear.

[36] 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.

[37] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170. ACM, New York, Jan. 1996.

[38] 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.

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

[40] D. Melski and T. Reps. Interconvertibility of set constraints and conext-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, June 1997.

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

[42] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[43] P. Mishra and U. Reddy. Declaration-free type checking. In *Conference Record of the 12th Annual ACM Symposium on POPL*, pages 7–21. ACM, New York, Jan. 1985.

[44] T. Mogensen. Separating binding times in language specifications. In FPCA 1989 [17], pages 12–25.

[45] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

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

[47] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 116–127. ACM, New York, June 1992.

[48] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, Vienna, Austria, July 1997.

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

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

[51] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, Jan. 1998.

[52] D. S. Scott. Lectures on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292. D. Reidel Publishing Company, 1982.

[53] H. Seidl and M. H. Sørensen. Constraints to stop deforestation. *Sci. Comput. Program.*, 32:73–107, 1998.

[54] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. ACM, New York, June 1988.

[55] M. H. Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *CAAP'94: Proceedings of the 19th International Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, Berlin, Apr. 1994.

[56] G. L. Steele. *Common Lisp the Language*. Digital Press, 1984.

[57] P. F. Sweeney and F. Tip. A study of dead data members in c++ applications. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 324–332. ACM, New York, June 1998.

[58] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[59] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73:231–248, 1990. Special issue of selected papers from the 2nd European Symposium on Programming.

[60] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, Berlin, Sept. 1987.

[61] M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 184–193. IEEE CS Press, Los Alamitos, Calif., May 1998.

[62] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Conference Record of the 26th Annual ACM Symposium on Principles of Programming Languages*, pages 291–302. ACM, New York, Jan. 1999.

[63] E. Wang and P. M. Hilfinger. Analysis of recursive types in lisp-like languages. In LFP 1992 [33], pages 216–225.

[64] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357, July 1984.

[65] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., 1993.