Eliminating Dead Computations on Recursive Data

Yanhong A. Liu*

July 1997

Abstract

This paper describes a general and effective method for backward dependence analysis in the presence of recursive data constructions. The goal is to identify partially dead recursive data and eliminate dead computations on them. The method uses projections based on general regular tree grammars extended with the notion of live and dead, and defines the analysis as mutually recursive grammar transformers. To guarantee that the analysis terminates, we describe how to use finite grammar abstract domains or carefully designed approximation operations. For methods based on finite domains, we describe a new application in caching intermediate results for program improvement. For methods based on approximation operations, we describe algorithms for three such operations that together produce more precise analysis results than previous methods. Finally, the analysis results are used to identify and eliminate dead computations.

1 Introduction

Dead computations produce values that never get used [3]. 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. There are also other programming activities that do not explicitly involve live or dead code but rely on similar notions, *e.g.*, program slices [42, 36]. Analysis for identifying live or dead code, or code of similar relevance, has been studied and used widely [9, 8, 22, 32, 3, 21, 18, 13, 23, 28, 27, 39, 36]. It is essentially backward dependence analysis that aims to compute minimal 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 that more dead code can be identified in more complicated computations [18, 13, 23, 27, 36, 6]. A particularly important problem is to identify dead computations on recursive data. Recursive data constructions are used increasingly widely in high-level languages, and dead code analysis for them is important for various program manipulations [21, 18, 28, 27, 36]. The goal is to identify *partially dead recursive data*.[1] It is difficult because the structures of general recursive data can be defined by the user, and dead computations on such data may be on recursive substructures interleaved with other recursive substructures on which the computations are live. Several methods have been studied, but all have limitations: specifying only heads and tails of list values [21], handling only list context where every

*This work is supported by a junior-faculty start-up grant from Indiana University. Author's address: Computer Science Department, Indiana University, 215 Lindley Hall, Bloomington, IN 47405. Phone: (812)855-4373. Email: liu@cs.indiana.edu.

[1]Note that this notion of partial deadness is different from the one in analysis of programs that do not use data constructions. There, partial dead code refers to code that is dead on some but not all computation paths.

head context and every tail context is the same [18], relying on crude methods to force termination [27], or using a limited class of regular tree grammars [36].

This paper describes a general and effective method for eliminating dead computations on recursive data. The method represents partially dead recursive data using projections based on general regular tree grammars extended with the notion of live and dead. The analysis is defined as mutually recursive grammar transformers that capture exact backward dependencies. To guarantee that the analysis terminates, we use either finite grammar abstract domains or carefully designed approximation operations. We show that appropriate finite grammar abstract domains can often be obtained for specific applications, and we describe a new application: caching intermediate results for program improvement. For general applications, we describe how to approximate argument projections with grammars that can be computed without iterating and how to approximate resulting projections with a widening operation. We design an approximation operation that combines two grammars efficiently to give the most precise deterministic result possible. The overall analysis yields more precise results than other known methods. Finally, the analysis results are used to identify and eliminate dead computations.

The rest of the paper is organized as follows. Section 2 describes a programming language with recursive data constructions. Section 3 discusses how to represent partially dead recursive data. Sections 4, 5, and 6 describe the analysis, finite grammar abstract domains, and approximation operations, respectively. Section 7 uses analysis results to eliminate dead computations. Section 8 discusses applications and relevant issues. Section 9 compares with related work and concludes.

2 Language

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

$e ::=$	v	variable
$\mid$	$c(e_1, ..., e_n)$	constructor application
$\mid$	$p(e_1, ..., e_n)$	primitive function application
$\mid$	$f(e_1, ..., e_n)$	function application
$\mid$	$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$	conditional expression
$\mid$	$\textbf{let } v = e_1 \textbf{ in } e_2$	binding expression

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 corresponding arities. When needed, we use c^n to denote that c has arity n. For each constructor c^n declared, there is a primitive function $c?$ that tests whether the argument is an application of c; for $n > 0$, there is a primitive function $c_{\bar{i}}$ for each $i = 1..n$ that selects the ith component in an application of c. A program is a set of mutually recursive function definitions of the form:

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

together with a set of constructor declarations, and a function f_0 that is to be evaluated with some input $x = \langle x_1, ..., x_n \rangle$. Figure 1 gives some example definitions, assuming that constructors nil^0 and $cons^2$ are declared in the programs where they are used. For ease of reading, we use car for $cons_{\bar{1}}$ and use cdr for $cons_{\bar{2}}$ in programs. Later we will also use constructor $triple^3$ and its corresponding selectors $1st$, $2nd$, and $3rd$.

We use call-by-value semantics for this language. Well-defined expressions evaluate to constructed data, such as $cons(3, nil)$, and we use $\perp$ to denote the value of undefined (non-terminating)

$odd(x)$: return elements of x at odd positions
$even(x)$: return elements of x at even positions

$$
\begin{aligned}
odd(x) &= \textbf{if } null(x) \textbf{ then } nil \\
&\quad \textbf{else } cons(car(x), even(cdr(x))) \\
even(x) &= \textbf{if } null(x) \textbf{ then } nil \\
&\quad \textbf{else } odd(cdr(x))
\end{aligned}
$$

$cut(n, l)$: for $n \geq 0$ and l longer than $2n+1$, take $2n+1$ elements off the head of l, then put n on the head

$$
\begin{aligned}
cut(n, l) &= \textbf{if } n = 0 \textbf{ then } cons(0, cdr(l)) \\
&\quad \textbf{else } cons(n, cdr(cdr(cut(n-1, cdr(l)))))
\end{aligned}
$$

Figure 1: Example function definitions

expressions. So, if the value of any subexpression is $\perp$, then the value of the super expression is $\perp$. For example, if the value of some e_i is $\perp$, then the value of $c(e_1, ..., e_n)$ is $\perp$. In particular, the tester and selectors corresponding to a constructor c are defined precisely as follows: given a value x,

$$
\begin{aligned}
c^n?(x) &= \begin{cases} \texttt{true} & \text{if } x = c^n(x_1, ..., x_n) \\ \texttt{false} & \text{if } x = d^m(x_1, ..., x_m) \text{ and } d \neq c \\ \perp & \text{otherwise} \end{cases} \\
c^n{}_{\bar{i}}(x) &= \begin{cases} x_i & \text{if } x = c^n(x_1, ..., x_n) \\ \perp & \text{otherwise} \end{cases}
\end{aligned} \tag{2}
$$

Since a program can use data constructions $c(e_1, ..., e_n)$ in recursive function definitions, the data it processes can be recursive. Therefore, we may have data structures of unbounded size, *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 any outputs of the program. To improve program efficiency, we can eliminate such dead computations and use a special symbol $\_$ as a placeholder for the value of such a computation. A constructor application might not evaluate to $\_$ even if some arguments evaluate to $\_$. A primitive function application or a conditional expression must evaluate to $\_$ (or $\perp$) if any of its subexpressions evaluate to $\_$. Whether a function application (or a binding expression) evaluates to $\_$ depends on the values of the arguments (or the bound variable) and how they are used in the function definition (or the body).

3 Representing partially dead recursive data

Projections. Since constructed data can be recursive, and subparts of them can be dead, to present analysis results, we first need a way of describing partially dead recursive data. Note that completely live or dead data are special cases of partially dead data.

Domain projection [37, 15] has been shown to be a clean tool for describing partial constructed data by simply projecting out the parts that are of interest [41, 24, 31, 36]; we use this notion to project out the parts of data that are live. Let X be the domain of all possible values computed by our programs, including $\perp$ and values containing $\_$. For any value x in X, $\perp \sqsubseteq x$; for two values x_1 and x_2 other than $\perp$,

$$
\begin{aligned}
x_1 \sqsubseteq x_2 \quad \text{iff} \quad & x_1 = \_, \text{ or} \\
& x_1 = x_2, \text{ or} \\
& x_1 = c(x_{11}, ..., x_{1n}),\ x_2 = c(x_{21}, ..., x_{2n}), \text{ and } x_{1i} \sqsubseteq x_{2i} \text{ for } i = 1..n.
\end{aligned} \tag{3}
$$

A *projection* over X is a function $\pi : X \to X$ such that $\pi(x) \sqsubseteq x$ and $\pi(\pi(x)) = \pi(x)$ for any $x \in X$. Two special projections are $I\!D$ and AB. $I\!D$ is the identity function: $I\!D(x) = x$. AB is the absence function: $AB(x) = \_$ for any $x \neq \perp$, and $AB(\perp) = \perp$. To project out parts of a constructed data $c^n(x_1, ..., x_n)$, we use projections denoted $T_{c^n}(\pi_1, ..., \pi_n)$, and we define:

$$T_{c^n}(\pi_1, ..., \pi_n)(x) = \begin{cases} c^n(\pi_1(x_1), ..., \pi_n(x_n)) & \text{if } x = c^n(x_1, ..., x_n) \\ \perp & \text{otherwise} \end{cases} \tag{4}$$

If a constructor c has arity 0, then we use T_c in place of $T_c()$. For example, $T_{cons}(I\!D, T_{cons}(AB, AB))$ projects out a *cons* structure with its head and with its tail projected out to a *cons* structure with neither the head nor the tail, and T_{nil} projects out simply the *nil* structure. To summarize, a projection is a function; when applied to constructed data, it returns the data with only the corresponding live parts left and the dead parts replaced by $\_$.

Grammar-based projections. Since our data structures can be recursive and thus have unbounded sizes, precise descriptions of partially dead data structures can have unbounded sizes as well so that they can not be computed exactly by a terminating analysis of programs.

Formal language theory is a powerful framework for representing unbounded structures using bounded representations, which are often even precise [12]. In particular, regular tree grammars have been used to describe partial data structures and other data flow information [20, 30, 31, 5, 38, 12, 36]. We describe partially dead recursive data using projections that are represented using regular tree grammars. A *regular-tree-grammar-based projection* G is a quadruple $\langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$, where $\mathcal{T}$ is a set of terminal symbols including $I\!D$, AB, and T_c for all constructors c, $\mathcal{N}$ is a set of nonterminal symbols N, $\mathcal{P}$ is a set of production rules of the form:

$$N \Rightarrow I\!D, \quad N \Rightarrow AB, \quad \text{or} \quad N \Rightarrow T_{c^n}(N_1, ..., N_n), \tag{5}$$

and S is a start symbol. The language L_G generated by G is the set $\{\pi \in \mathcal{T}^* \mid S \overset{*}{\Rightarrow}_G \pi\}$ of sentences. The projection function that G represents is:

$$G(x) = \sqcup\{\pi(x) \mid \pi \in L_G\}. \tag{6}$$

where $\sqcup$ is the least upper bound operation for $\sqsubseteq$. It's easy to see that $G(x)$ is well-defined for $x \in X$. In particular, we overload $I\!D$ to denote the grammar with only a production $S \Rightarrow I\!D$, and overload AB to denote the grammar with only a production $S \Rightarrow AB$. For ease of presentation, when no confusion arises, we use the start symbol S, with associated productions when necessary, possibly in compact forms, to denote such a grammar-based projection. For example, $\{S \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$ [2] projects out a list with only its spine but not any elements, and $\{S \Rightarrow T_{nil} \mid T_{cons}(N, S), N \Rightarrow ...\}$ projects out a list whose elements are projected by N.

For convenience of the analysis framework, we extend the regular tree grammars to allow productions of the form:

$$N \Rightarrow N' \quad \text{or} \quad N \Rightarrow T_{c_i^-}(N'), \tag{7}$$

where terminal symbols $T_{c_i^-}$ are for selectors c_i^-, and we define:

$$T_{c_i^-}(\pi) = \begin{cases} I\!D & \text{if } \pi = I\!D \\ \pi_i & \text{if } \pi = T_{c^n}(\pi_1, ..., \pi_n) \\ AB & \text{otherwise} \end{cases} \tag{8}$$

[2]Note that this denotes a grammar with one production in compact form. It does not denote a set comprehension.

Given such an extended regular tree grammar G whose production set $\mathcal{P}$ contains productions of the form (5) and (7), we can define a relation $\stackrel{s}{\Rightarrow}$ to shortcut productions of the form (7):

$$N \stackrel{s}{\Rightarrow} R \quad \text{iff} \quad \begin{array}{l} N \Rightarrow R, \text{ where } R \text{ is of the form of a right hand side of (5), or} \\ N \Rightarrow N' \text{ and } N' \stackrel{s}{\Rightarrow} R, \text{ or} \\ N \Rightarrow T_{c_i^{-}}(N'),\ N' \stackrel{s}{\Rightarrow} T_{c^n}(N_1, ..., N_n), \text{ and } N_i \stackrel{s}{\Rightarrow} R\,. \end{array} \tag{9}$$

We can construct a regular tree grammar G' whose production set $\mathcal{P}'$ contains only productions of the form (5) such that $L_G = L_{G'}$:

$$\mathcal{P}' = \mathcal{P} \cup \{N \Rightarrow R : N \stackrel{s}{\Rightarrow} R\} - \{N \Rightarrow Z : Z \text{ is of the form of a right hand side of (7)}\}. \tag{10}$$

This construction is similar to when only the selector form is used [20]. The proof that $L_G = L_{G'}$ is also similar, and we omit it here. We simply call an extended regular tree grammar a regular tree grammar.

Grammar abstract domains. Program analysis associates abstract values, such as grammar-based projections, with particular indices, such as functions, parameters, and subexpressions. A projection associated with an index indicates how much of the value computed at this index is live.

Let $\mathcal{I}$ be the set of indices I, and $\mathcal{G}$ be the set of regular tree grammars G. Define an abstract domain $\mathcal{D} = \mathcal{I} \rightarrow \mathcal{G}$. For two regular-tree-grammar-based projections G_1 and G_2, we define:

$$G_1 \leq G_2 \quad \text{iff} \quad \forall \pi_1 \in L_{G_1},\ \exists \pi_2 \in L_{G_2},\ \pi_1 \leq \pi_2, \tag{11}$$

where for two sentences π_1 and π_2, we define by overloading $\leq$:

$$\pi_1 \leq \pi_2 \quad \text{iff} \quad \begin{array}{l} \pi_1 = AB, \text{ or} \\ \pi_2 = ID, \text{ or} \\ \pi_1 = T_c(\pi_{11}, ..., \pi_{1n}),\ \pi_2 = T_c(\pi_{21}, ..., \pi_{2n}), \text{ and } \pi_{1i} \leq \pi_{2i} \text{ for } i = 1..n. \end{array} \tag{12}$$

We can compute the *least upper bound* $G_1 \vee G_2$ of two grammars $G_1 = \langle \mathcal{T}_1, \mathcal{N}_1, \mathcal{P}_1, S_1 \rangle$ and $G_2 = \langle \mathcal{T}_2, \mathcal{N}_2, \mathcal{P}_2, S_2 \rangle$, assuming all nonterminals in them are renamed so that those in G_1 are distinct from those in G_2, and S is a nonterminal not used in G_1 or G_2:

$$G_1 \vee G_2 = \langle \mathcal{T}_1 \cup \mathcal{T}_2,\ \mathcal{N}_1 \cup \mathcal{N}_2 \cup \{S\},\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \Rightarrow S_1, S \Rightarrow S_2\},\ S \rangle \tag{13}$$

We say two grammars G_1 and G_2 are equivalent iff $G_1 \leq G_2$ and $G_2 \leq G_1$. We reason about grammar orderings modulo this equivalence. This equivalence is decidable for two reasons. First, given G_1 and G_2, the question $L_{G_1} \subseteq L_{G_2}$ is decidable [14, 4]. Second, given G_1 and G_2, we can construct G_1' and G_2' such that $G_1 \leq G_2$ iff $L_{G_1'} \subseteq L_{G_2'}$. The construction of G_i' (for $i = 1, 2$) from G_1 and G_2, has three steps, assuming that nonterminals in them are renamed so that those in G_1 are distinct from those in G_2:

1. Let $G_i' = G_i$.
2. If $N \Rightarrow ID \in \mathcal{P}_i$, then add to $\mathcal{P}_i'$ the production $N \Rightarrow AB$ and the productions $N \Rightarrow T_{c^n}(N_1, ..., N_n)$ for all terminals T_{c^n} and nonterminals $N_1, ..., N_n$ used in $\mathcal{P}_1$ or $\mathcal{P}_2$.
3. If $N \Rightarrow T_{c^n}(N_1, ..., N_n) \in \mathcal{P}_i$ for any terminal T_{c^n} and nonterminals $N_1, ..., N_n$, then add to $\mathcal{P}_i'$ the production $N \Rightarrow AB$.

Step 2 guarantees that, if N can derive $I\!D$, then N can derive every sentence in L_{G_1} and L_{G_2}. Step 3 guarantees that, if N can derive some sentence, then it can derive AB. The above grammar ordering can be defined in a pointwise fashion (with respect to elements of $\mathcal{I}$) on the domain $\mathcal{D}$, though we will mostly use the ordering on a per-point basis. Note that this (pointwise) ordering does not form a complete partial order.

We use CON to denote the grammar with productions $S \Rightarrow T_{c^n}(\overbrace{N,...,N}^{n})$ for all possible constructors c^n and $N \Rightarrow AB$. Given a grammar $G = \langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$, we use $G_{c_{\bar{i}}}$ to denote the part of G restricted to the ith component of a c^n structure ($i \leq n$), assuming S_i is a nonterminal not in used in G:

$$G_{c_{\bar{i}}} = \langle \mathcal{T},\ \mathcal{N} \cup \{S_i\},\ \mathcal{P} \cup \{S_i \Rightarrow T_{c_{\bar{i}}}(S)\},\ S_i \rangle \tag{14}$$

and we use G_{c_i} to denote using G as the ith component of a c^n structure ($i \leq n$), assuming that S_0 is a nonterminal not used in G:

$$G_{c_i} = \langle \mathcal{T},\ \mathcal{N} \cup \{S_0\},\ \mathcal{P} \cup \{S_0 \Rightarrow T_{c^n}(\overbrace{N,...,N}^{i-1}, S, \overbrace{N,...,N}^{n-i}), N \Rightarrow AB\},\ S_0 \rangle \tag{15}$$

For example, if nil and $cons$ are all possible constructors in values computed in a program, as we assume for functions odd and $even$, then $CON = \{S \Rightarrow T_{nil} \mid T_{cons}(AB, AB)\}$. If $G = I\!D$, then $G_{cons_{\bar{1}}} = G_{cons_{\bar{2}}} = I\!D$, $G_{cons_1} = \{S \Rightarrow T_{cons}(I\!D, AB)\}$, and $G_{cons_2} = \{S \Rightarrow T_{cons}(AB, I\!D)\}$.

4 Analyzing partially dead recursive data

We develop a backward analysis that, given how much of the value of a function f is live, computes how much of each parameter of f is live.

Transformers for grammar-based projections. The basic idea is that a projection associated with an index can be transformed into projections associated other indices based on the semantics of the program segments involved. For example, how much of result of a function is live can be transformed into how much of a parameter is live. We use f^i to denote such a *projection transformer*: it takes a projection that is applied to the result of f and returns a projection that is to be applied to the ith parameter; f^i must satisfy the *sufficiency condition*: if $G_i = f^i(G)$, then

$$G(f(v_1, ..., v_i, ..., v_n)) \sqsubseteq f(v_1, ..., G_i(v_i), ..., v_n) \tag{16}$$

for all values of $v_1, ..., v_n$. Similarly, how much of the value of an expression is live can be transformed into how much of a variable is live. We use e^v to denote such a projection transformer: it takes a projection that is applied to e and returns a projection that is to be applied to every instance of v in e; a similar sufficiency condition must be satisfied: if $G' = e^v(G)$, then

$$G(e) \sqsubseteq e[G'(v)/v] \tag{17}$$

for all values of the variables in e.

For a function definition of the form $f(v_1, ..., v_n) = e$, how much of the ith parameter is live in computing the value of f is just how much of v_i is live in computing the value of e. So, we define $f^i(G) = e^{v_i}(G)$ for each f and i, and define e^v based on the structure of e, possibly referring to the f^i's, thus forming recursive definitions. We define:

$$e^v(AB) = AB. \tag{18}$$

For $G \neq AB$, we define $e^v(G)$ in Figure 2. Rules (1) and (2) are straightforward. Rules (3)-(5) handle data constructions; as a special case of (3), for a constructor c of 0 arity, the right hand side is AB. Rule (6) is simple; note that, if we assume that these primitive functions, not including c_i^-'s or $c?$'s, are defined on only primitive types such as boolean or integer, then we can use CON's in place of ID's in the right hand side. Rule (7) is the recursive definition using f^i's. Rules (8) follows from the semantics of conditional expressions; in fact, we can use CON in place of ID as a sufficient context for the condition. Rule (9) follows from the semantics of binding expressions; it assumes that the bound variable u has been renamed so that it is different from v.

(1)	$v^v(G)$	$=$	G	
(2)	$u^v(G)$	$=$	AB	if $u \neq v$
(3)	$(c(e_1, ..., e_n))^v(G)$	$=$	$e_1{}^v(G_{c_1^-}) \ \vee \ ... \ \vee \ e_n{}^v(G_{c_n^-})$	
(4)	$(c_i^-(e))^v(G)$	$=$	$e^v(G_{c_i})$	
(5)	$(c?(e))^v(G)$	$=$	$e^v(CON)$	
(6)	$(q(e_1, ..., e_n))^v(G)$	$=$	$e_1{}^v(ID) \ \vee \ ... \ \vee \ e_n{}^v(ID)$	if q is p other than c_i^- or $c?$
(7)	$(f(e_1, ..., e_n))^v(G)$	$=$	$e_1{}^v(f^1(G)) \ \vee \ ... \ \vee \ e_n{}^v(f^n(G))$	
(8)	$(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^v(G)$	$=$	$e_1{}^v(ID) \ \vee \ e_2{}^v(G) \ \vee \ e_3{}^v(G)$	
(9)	$(\textbf{let } u{=}e_1 \textbf{ in } e_2)^v(G)$	$=$	$e_1{}^v(e_2{}^u(G)) \ \vee \ e_2{}^v(G)$	assume $u \neq v$ after renaming

Figure 2: Definition of $e^v(G)$ for $G \neq AB$

It is easy to see that each rule guarantees sufficient information, and thus the sufficiency conditions are satisfied by recursion induction. In particular, our analysis does not report error, *i.e.*, undefined values. For example, in rule (3), if the argument G contains no sentence of the form $T_c(\pi_1, ..., \pi_n)$, then the program contains an unexpected error; by definition, $G_{c_i^-} = AB$ for all $i = 1..n$. Also, in rule (5), if the argument is not AB or ID, then the program contains an unexpected error. This does not falsify our sufficiency conditions since, where the program contains an error, the left hand side of a sufficiency condition is $\perp$ and thus the condition holds definitely. On the other hand, we could extend our analysis to report the above errors by adding a top projection $TOP(x) = \perp$ that denotes and propagates them but, due to undecidabilities, no analysis can report all errors. So, we have kept our analysis simple and sound. We can also easily show by induction that each transformer is a monotone function.

Example 4.1 For the functions *odd*, *even*, and *cut* in Figure 1, we obtain the following definitions:

$$\begin{array}{lll}
odd^1(G) & = (null(x))^x(ID) \vee nil^x(G) \vee (cons(car(x), even(cdr(x))))^x(G) & \text{by (8)} \\
& = x^x(CON) \vee AB \vee (car(x))^x(G_{cons_1^-}) \vee (even(cdr(x)))^x(G_{cons_2^-}) & \text{by (5)(3)(3)} \\
& = CON \vee AB \vee x^x(G_{cons_1^-})_{cons_1} \vee (cdr(x))^x(even^1(G_{cons_2^-})) & \text{by (1)(4)(7)} \\
& = CON \vee AB \vee (G_{cons_1^-})_{cons_1} \vee (even^1(G_{cons_2^-}))_{cons_2} & \text{by (1)(4)(1)} \\
& = CON \vee (G_{cons_1^-})_{cons_1} \vee (even^1(G_{cons_2^-}))_{cons_2} & \text{simplification} \\
even^1(G) & = CON \vee (odd^1(G))_{cons_2} & \text{similarly} \\
cut^2(G) & = (G_{cons_2^-})_{cons_2} \vee (cut^2(((G_{cons_2^-})_{cons_2})_{cons_2}))_{cons_2} &
\end{array}$$

For example, $odd^1(ID)$ computes how much of the (first) argument of *odd* is live, $cut^2(ID)$ computes how much of the second argument of *cut* is live, and $cut^2(\{S \Rightarrow T_{cons}(AB, AB)\})$ computes how much of the second argument of *cut* is needed to return a *cons* structure.

Computing transformer applications. To compute $f_0^{i_0}(G_0)$ for some $f_0^{i_0}$ and G_0, if the definition of $f_0^{i_0}$ does not involve recursion, then we can compute directly using its definition. If the definition involves recursion, then we can start at the bottom element for every f^i, *i.e.*, $f^{i(0)} = \lambda G.AB$, and iteratively compute $f^{i(k+1)}$'s using the values of $f^{i(k)}$'s until all involved values stabilize. Here, $f^{i(k+1)}$ is the approximation in iteration $k+1$ to the value of f^i. However, this iteration may not terminate, for two reasons.

First, computing certain $f_1^{i_1(k_1)}(G_1)$ may involve computing $f_1^{i_1(k_1-1)}(G_2)$ for a new G_2, and $f_1^{i_1}(G_2)$ must stabilize before $f_1^{i_1}(G_1)$ can, but then similarly we may need to first stabilize $f_1^{i_1}(G_3)$, $f_1^{i_1}(G_4)$, and so on, and this may never terminate. For example, to compute $cut^2(ID)$, we need to compute $cut^2(\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, ID))\})$, $cut^2(\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons}(AB, ID)))\})$, and so on, even though we can see that the result seems to stabilize at $\{S \Rightarrow T_{cons}(AB, ID)\}$. See the table below. In this table, each column contains successive approximations to the value of applying cut^2 to the projection at the top of the column. To help track the results of different transformer applications, we attach a unique tag, shown as a subscript, for the initial value AB of each transformer application and keep the tag as long as possible.

cut^2	ID	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, ID))\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons}(AB, ID)))\}$	...
(0)	AB_1	AB_2	AB_3	...
(1)	$\{S \Rightarrow T_{cons}(AB, N),$ $N \Rightarrow ID \mid AB_2\}$ simplify: $\{S \Rightarrow T_{cons}(AB, ID)\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, N_1)),$ $N_1 \Rightarrow ID \mid AB_3\}$ simplify: $\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, ID))\}$	...	
(2)	$\{S \Rightarrow T_{cons}(AB, N_2),$ $N_2 \Rightarrow ID \mid T_{cons}(AB, T_{cons}(AB, ID))\}$ simplify: $\{S \Rightarrow T_{cons}(AB, ID)\}$	...		
...	...			

For another example, to compute $cut^2(\{S \Rightarrow T_{cons}(AB, AB)\})$, we need to compute $cut^2(\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, AB))\})$, $cut^2(\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons}(AB, AB)))\})$, and so on, and the result does not even stabilize:

cut^2	$\{S \Rightarrow T_{cons}(AB, AB)\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, AB))\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons}(AB, AB)))\}$	...
(0)	AB_1	AB_2	AB_3	AB_4
(1)	$\{S \Rightarrow T_{cons}(AB, N),$ $N \Rightarrow AB \mid AB_2\}$ simplify: $\{S \Rightarrow T_{cons}(AB, AB)\}$	$\{S \Rightarrow T_{cons}(AB, N),$ $N \Rightarrow T_{cons}(AB, AB) \mid AB_3\}$ simplify: $\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, AB))\}$	$\{S \Rightarrow T_{cons}(AB, N_1),$ $N_1 \Rightarrow T_{cons}(AB, T_{cons}(AB, AB)) \mid AB_4\}$ simplify: $\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons}(AB, AB)))\}$	
(2)	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, N)),$ $N \Rightarrow T_{cons}(AB, AB) \mid AB_3\}$ simplify: $\{S \Rightarrow T_{cons}(AB, T_{cons}(AB,$ $T_{cons}(AB, AB)))\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, N_1)),$ $N_1 \Rightarrow T_{cons}(AB, T_{cons}(AB, AB)) \mid AB_4\}$ simplify: $\{S \Rightarrow T_{cons}(AB, T_{cons}(AB,$ $T_{cons}(AB, T_{cons}(AB, AB))))\}$		
...	...			

Second, even if $f_1^{i_1}$ needs to be computed on a finite number of arguments, the resulting projections of, say, $f_1^{i_1}(G_1)$ may be a strictly increasing chain of grammars with no limit. For example, the resulting projections of $odd^1(ID)$ and $even^1(ID)$ are strictly increasing:

	$odd^1(ID)$	$even^1(ID)$
(0)	AB_o	AB_e
(1)	$\{S \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$	$\{S \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o)\}$
(2)	$\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N),\ N \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o)\}$	$\{S \Rightarrow T_{nil} \mid T_{cons}(AB, M),\ M \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$
...	...	...

There are three standard ways to guarantee the termination of the iteration: using finite transformers, using finite abstract domains, or using approximation operations. Finite transformers can be obtained by restricting them to be written in a specific meta-language [12]. This meta-language corresponds to a restricted class of regular tree grammars extended with selectors [20, 12] and can be rewritten as a set of constraints [16, 17, 12]. As discussed by Cousot and Cousot, this is essentially a masking of the explicit use of an approximation, called widening, when defining transformers [12]. Appropriate finite abstract domains can often be obtained for various applications of the analysis, and they can provide sufficiently precise analysis results on a per-program basis. We discuss this method and one of its new applications in Section 5. Approximation operations provide a more general solution and make the analysis framework more modular and flexible. In Section 6, we describe three approximation operations that together allow our analysis to give more precise results than previous methods.

5 Using finite grammar abstract domains

As with most analyses based on abstract interpretation [1], we can use a finite abstract domain $\mathcal{D}$. In particular, we can do this on a per-program basis. Since the set $\mathcal{I}$ of indices is finite for a given program, we just need to use a finite domain $\mathcal{G}$ of grammars.

Once $\mathcal{G}$ is finite, to compute $f_0^{i_0}(G_0)$ for some $f_0^{i_0}$ and G_0, we can start at $f^i(G) = AB$ for every f^i and G in the finite domain of grammars and iterate using the definitions until a fixed point is reached, *i.e.*, we set $f^{i(0)}(G) = AB$ for all f^i and G, and we compute $f^{i(k+1)}(G)$'s following their definitions, using the values of $f^{i(k)}(G)$'s when needed, until their values stabilize. This always terminates since the abstract domain is finite. Finally, we can retrieve the value of $f_0^{i_0}(G_0)$ for the particular $f_0^{i_0}$ and G_0 of interest.

We can also compute $f_0^{i_0}(G_0)$ for particular $f_0^{i_0}$ and G_0 in an *on-demand* fashion. We demand to compute $f_0^{i_0(0)}(G_0), f_0^{i_0(1)}(G_0)$, and so on, one in each iteration. If, during the computation of $f_0^{i_0(k)}(G_0)$, the value of $f_1^{i_1(k_1)}(G_1)$ is needed but is not computed yet, then we recursively demand to compute it; note that $k_1 = k - 1$, *i.e.*, the computation always uses values from the previous iteration, and thus the recursion occurs at most k times. The overall iteration stops when the values of all $f^i(G)$'s involved stabilize. On-demand evaluation is usually more efficient since it computes only values needed in computing $f_0^{i_0}(G_0)$. It is also more general, since it can be used when a finite abstract domain is not given a priori but certain approximation operations are used to guarantee termination, as we will see in Section 6.

A simple method. To make $\mathcal{G}$ finite, we can simply restrict the set of nonterminals to include only those that denote projections associated with indices for the given program. This makes the set of nonterminal symbols finite. For any given program, the set of terminal symbols is also finite. Thus, the number of productions of the form (5) or (7) using these symbols is finite also. So $\mathcal{G}$ is finite. The set of indices should be selected to minimize analysis effort while preserving precise analysis results. For example, it could be the set of function parameters, corresponding to the projection transformers f^i.

We describe a way of implementing such a restriction in our analysis. We associate a unique nonterminal N with each selected index, and we force that the projection associate with such an index use N as the start symbol and that this projection be referred through N in all uses. If a projection being computed is not associated with a selected index, we use a temporary nonterminal as its start symbol, but we force the symbol to be eliminated in the projection that immediately uses it and is associated with a selected index. This in effect forces projections obtained from different iterations to be combined, possibly approximately, thereby avoiding introduction of an unbounded number of new nonterminals and ensuring termination.

Example 5.1 For the *odd* and *even* example, if we associate nonterminal N_1 with the result of odd^1, associate N_2 with the result of $even^1$, and use N_1 and N_2 as the respective start symbols, we reach fixed points after three iterations:

	$odd^1(ID)$	$even^1(ID)$
(0)	$\{N_1 \Rightarrow AB_o\}$	$\{N_2 \Rightarrow AB_e\}$
(1)	$\{N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2),\ N_2 \Rightarrow AB_e\}$	$\{N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1),\ N_1 \Rightarrow AB_o\}$
(2)	$\{N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2),\ N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1)\}$	$\{N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1),\ N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2)\}$
(3)	$\{N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2),\ N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1)\}$	$\{N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1),\ N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2)\}$

We could associate additional nonterminals with other indices, but that would lead to the same result. For the cut^2 example, we can associate a nonterminal N with the result of cut^2, associate M with the argument of cut^2, and obtain:

cut^2	$\{M \Rightarrow ID\}$	$\{M \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons\bar{2}}(M))),\ M \Rightarrow ID\} = \{M \Rightarrow ID\}$
(0)	$\{N \Rightarrow AB_1\}$	same as left
(1)	$\{N \Rightarrow T_{cons}(AB, ID)\}$	same as left
(2)	$\{N \Rightarrow T_{cons}(AB, ID)\}$	same as left

cut^2	$\{M \Rightarrow T_{cons}(AB, AB)\}$	$\{M \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons\bar{2}}(M))),\ M \Rightarrow T_{cons}(AB, AB)\}$
(0)	$\{N \Rightarrow AB_1\}$	$\{N \Rightarrow AB_2\}$
(1)	$\{N \Rightarrow AB \mid T_{cons}(AB, N)\}$	$\{N \Rightarrow AB \mid T_{cons}(AB, N)\}$
(2)	$\{N \Rightarrow AB \mid T_{cons}(AB, N)\}$	$\{N \Rightarrow AB \mid T_{cons}(AB, N)\}$

Application-based methods. Even though the above simple method always works, we may develop appropriate finite grammar abstract domains for particular applications.

As a special example, an incremental program $\bar{f}'(x, \delta x, \bar{r})$ may use values embedded in $\bar{r}$, a tree of all intermediate results of a previous computation $\bar{f}(x)$ [27]. In particular, $\bar{r}$ is obtained from the definition of $\bar{f}$ and has a finite structure. This finite structure is used to define a finite grammar abstract domain. Thus, we can use the above analysis to compute $\bar{f}'^3$ and determine how much of $\bar{r}$ is needed in the incremental computation. Using this information, we can prune $\bar{f}$ so that it returns only intermediate results that are useful in the incremental computation. This problem is important in the cache-and-prune method for program improvement [27]. We illustrate this method with an example.

Example 5.2 Binomial coefficient program $b(i,j)$ computes the number of j-element subsets of an i-element set for $0 \le j \le i$, and it takes exponential time:

$$\begin{array}{ll} b(i,j) = & \textbf{if } j = 0\ \vee j = i \textbf{ then } 1 \\ & \textbf{else } b(i-1, j-1) + b(i-1, j) \end{array}$$

We want to obtain a program that efficiently computes $b(i+1, j)$ by using $b(i,j)$. Using the cache-and-prune method [27], we (i) cache all intermediate results of b and obtain $\bar{b}$, so $b = 1st(\bar{b}(i,j))$, and (ii) incrementalize $\bar{b}$ and obtain $\bar{b}'$, so $\bar{b}'(i, j, \bar{r}) = \bar{b}(i+1, j)$ for $\bar{r} = \bar{b}(i,j)$:

$$\begin{array}{l} \bar{b}(i,j) \\ = \ \textbf{if } j = 0\ \vee j = i \textbf{ then } triple(1, _, _) \\ \quad \textbf{else let } v = \bar{b}(i-1, j-1), \\ \qquad\qquad u = \bar{b}(i-1, j) \\ \qquad \textbf{in } triple(1st(v) + 1st(u), v, u) \end{array}
\qquad
\begin{array}{l} \bar{b}'(i, j, \bar{r}) \\ = \ \textbf{if } j = 0\ \vee j = i+1 \textbf{ then } triple(1, _, _) \\ \quad \textbf{else if } j = i \textbf{ then} \\ \qquad \textbf{let } v = \bar{b}'(i-1, j-1, triple(1, _, _)) \\ \qquad \textbf{in } triple(1st(v) + 1, v, triple(1, _, _)) \\ \quad \textbf{else let } v = \bar{b}'(i-1, j-1, 2nd(\bar{r})) \\ \qquad \textbf{in } triple(1st(v) + 1st(\bar{r}), v, \bar{r}) \end{array}$$

Note that $\bar{b}'(i,j,\bar{r})$ takes $O(j)$ time but $\bar{r}$ takes exponential space. Using the methods in this paper, we can prune unnecessary recursive data in $\bar{r}$ and reduce its space consumption to $O(j)$.

First, from the definition of $\bar{b}$, we can directly represent the finite structure of $\bar{r}$ using a grammar

$$G_{\bar{b}} = \{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, S)\}.$$

Since the first component of an application of *triple* contains the result of an application of b, ID in the grammar $G_{\bar{b}}$ is the projection applied to the result, indicating that the result is always needed. We restrict to a finite domain of grammars where each grammar is obtained from $G_{\bar{b}}$ by replacing uses of nonterminals in the right hand sides with AB's; in this example, we obtain the following four grammars:

$$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, S)\}, \quad \{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, AB, S)\},$$
$$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}, \; \{S \Rightarrow T_{triple}(ID, AB, AB)\}.$$

Then, for incremental computation, we need only parts of $\bar{r}$, in particular, the closure of the dependencies computed from $G_0 = \{S \Rightarrow T_{triple}(ID, AB, AB)\}$ and $G_{k+1} = \bar{b}'^3(G_k)$ [27]. So, we define $\bar{b}'^3$, which specifies how much of $\bar{r}$ is needed in computing $\bar{b}'(i,j,\bar{r})$:

$$\bar{b}'^3(G) = (\bar{b}'^3((G_{triple_{\bar{1}}})_{triple_1} \vee G_{triple_{\bar{2}}}))_{triple_2} \vee (G_{triple_{\bar{1}}})_{triple_1} \vee G_{triple_{\bar{3}}}.$$

Finally, we compute $\bar{b}'^3(G_0)$ and reach a fixed point $G_1 = \{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}$, and we compute $\bar{b}'^3(G_1)$ and reach the same fixed point. So, G_1 is the closure.

	$\bar{b}'^3(\{S \Rightarrow T_{triple}(ID, AB, AB)\})$
(0)	AB_1
(1)	$\{S \Rightarrow T_{triple}(ID, AB_1, AB)\}$
(2)	$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}$
(3)	$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}$

	$\bar{b}'^3(\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\})$
(0)	AB_1
(1)	$\{S \Rightarrow T_{triple}(ID, AB_1, AB)\}$
(2)	$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}$
(3)	$\{S \Rightarrow T_{triple}(ID, AB, AB) \mid T_{triple}(ID, S, AB)\}$

As we will see in Section 7, according to G_1, we can simply replace the third component of a *triple* constructor with _; this is done in the second and third branch of $\bar{b}'$. We obtain an incremental program that takes $O(j)$ time and $O(j)$ space:

$$\begin{array}{ll} & \bar{b}'(i,j,\bar{r}) \\ = & \textbf{if } j=0 \;\vee j=i+1 \textbf{ then } triple(1,\_,\_) \\ & \quad \textbf{else if } j=i \textbf{ then} \\ & \quad\quad \textbf{let } v=\bar{b}'(i-1,j-1,triple(1,\_,\_)) \\ & \quad\quad \textbf{in } \; triple(1st(v)+1,v,\_) \\ & \quad \textbf{else let } v=\bar{b}'(i-1,j-1,2nd(\bar{r})) \\ & \quad\quad \textbf{in } \; triple(1st(v)+1st(\bar{r}),v,\_) \end{array}$$

Furthermore, this efficient incremental program can also be used to form a new program $b_1(i,j)$ that computes $b(i,j)$ in $O(i*j)$ time and $O(j)$ space, which are asymptotically optimal:

$$b_1(i,j) = 1st(\bar{b}_1(i,j)) \qquad \bar{b}_1(i,j) = \begin{array}{l} \textbf{if } j=0 \;\vee j=i \textbf{ then } triple(1,\_,\_) \\ \textbf{else let } \bar{r}=\bar{b}_1(i-1,j) \textbf{ in } \bar{b}'(i-1,j,\bar{r}) \end{array}$$

6 Approximation operations

We describe three approximation operations. The first one guarantees that each transformer needs to be applied on only a finite number of arguments. The second one guarantees that the results of all transformer applications stabilize after a finite number of iterations. The third one combines two grammars to give the most precise deterministic grammar.

Approximating the argument projections. The goal is to guarantee that each transformer f^i needs to be computed on only a finite number of argument projections. We compute $f_0^{i_0}(G_0)$ iteratively and in an on-demand fashion. For each transformer f^i, we keep a list of all arguments on which it has been called. In computing $f_0^{i_0(k)}(G_0)$ for each k, we keep a stack of all transformer applications needed. If, while some $f_1^{i_1(k_1)}(G_1)$ is in the stack, we need to recursively call $f_1^{i_1(k_1-1)}(G_2)$ for a new argument G_2 to $f_1^{i_1}$, then we need to make sure that $f_1^{i_1}$ does not run into infinitely many new arguments. Note that, since G_2 is a new argument, it must be that $k_1 = 1$. As a safe case, if $size(G_2) < size(G_1)$ for a given well-founded measure $size$, then we continue the normal on-demand evaluation. Since each regular tree grammar corresponds to a finite automata, which can be minimized [14], we can use the number of states as the measure.

If $size(G_2) \not< size(G_1)$, then we obtain, symbolically, from G_2 and the definition of $f_1^{i_1}(G)$, a grammar G_2' that is a sufficient approximation of G_2 and possible future arguments that can be obtained following the pattern of this recursive call, and we use $f_1^{i_1(k_1-1)}(G_2')$ in place of $f_1^{i_1(k_1-1)}(G_2)$. We construct G_2' as follows. First, let S denote the start symbol of G in the definition of $f_1^{i_1}(G)$, and let S' denote the start symbol of G' in the recursive call $f_1^{i_1}(G')$ that corresponds to the call $f_1^{i_1(k_1-1)}(G_2)$. Then, represent S' symbolically in terms of S by following the definition of $f_1^{i_1}(G)$ syntactically, which possibly involves other transformer applications in between, and obtain $\{S' \Rightarrow ...,\ ...,\ ... \Rightarrow ...S...\}$. Finally, let S_2 denote the start symbol of G_2, and define $G_2' = \{S \Rightarrow S_2,\ S \Rightarrow S'\}$, where the productions for S_2 and S' are as in G_2 and G', respectively.

We have $G_2 \leq G_2'$ and thus $f_1^{i_1(k_1-1)}(G_2) \leq f_1^{i_1(k_1-1)}(G_2')$. Therefore, the sufficiency conditions still hold. Also, in computing $f_1^{i_1(k_1-1)}(G_2')$, the corresponding recursive call to $f_1^{i_1}$ will have the same argument G_2', and thus there can no longer be growth of argument projections of this pattern. In particular, if G_2 and possible future arguments form an increasing chain, then G_2' is the least upper bound; if G_2 and possible future arguments form a decreasing chain, then G_2' is a conservative approximation that equals G_2 but is greater than the rest.

Actually, we can let the approximation start from G_1 instead of G_2, *i.e.*, let S_1 denote the start symbol of G_1, define $G_1' = \{S \Rightarrow S_1, S \Rightarrow S'\}$, and restart to compute $f_1^{i_1(k_1-1)}(G_1')$ in place of $f_1^{i_1(k_1-1)}(G_1)$. This is easy since $k_1 = 1$. This actually makes the manipulation easier since G_1' avoids one round of grammar unfolding that resulted in G_2'.

Example 6.1 Consider the transformer cut^2. If S is the start symbol for G, and S' is the start symbol for $G' = ((G_{cons_{\bar{2}}})_{cons_2})_{cons_2}$, then

$$S' \Rightarrow T_{cons}(AB, M),\ M \Rightarrow T_{cons}(AB, N),\ N \Rightarrow T_{cons_{\bar{2}}}(S),\quad i.e.,\quad S' \Rightarrow T_{cons}(AB, T_{cons}(AB, T_{cons_{\bar{2}}}(S))).$$

In computing $cut^2(I\!D)$, we have $G_1 = I\!D$, and we obtain $G_1' = \{S \Rightarrow I\!D,\ S \Rightarrow S'\} = \{S \Rightarrow I\!D\}$. It reaches a fixed point after two iterations:

cut^2	$I\!D$	G_1'
(0)	AB_1	same as left
(1)	$\{S \Rightarrow T_{cons}(AB, N),\ N \Rightarrow I\!D \mid AB_1\}$ simplify: $\{S \Rightarrow T_{cons}(AB, I\!D)\}$	same as left
(2)	$\{S \Rightarrow T_{cons}(AB, N_1),\ N_1 \Rightarrow I\!D \mid T_{cons}(AB, N),\ N \Rightarrow I\!D \mid AB_1\}$ simplify: $\{S \Rightarrow T_{cons}(AB, I\!D)\}$	same as left

In computing $cut^2(\{S \Rightarrow T_{cons}(AB, AB)\})$, we have $G_1 = \{S \Rightarrow T_{cons}(AB, AB)\}$, and we obtain $G_1' = \{S \Rightarrow T_{cons}(AB, AB),\ S \Rightarrow S'\} = \{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, T_{cons}(AB, T_{cons_{\bar{2}}}(S)))\} = \{S \Rightarrow$

$T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$. We see that $cut^2(G_1')$ reaches a fixed point after two iterations:

cut^2	$\{S \Rightarrow T_{cons}(AB, AB)\}$	G_1'
$^{(0)}$	AB_1	AB_2
$^{(1)}$	$\{S \Rightarrow T_{cons}(AB, AB_2)\}$ simplify: $\{S \Rightarrow T_{cons}(AB, AB)\}$	$\{S \Rightarrow T_{cons}(AB, N),\ N \Rightarrow T_{cons}(AB, N) \mid AB_2\}$ simplify: $\{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$
$^{(2)}$	$\{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$	$\{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$

Approximating the resulting projections after each iteration. To guarantee that all $f^i(G)$'s involved stabilize after a finite number of iterations, we use a widening operation. The idea of widening was first proposed by Cousot and Cousot [11]. A widening operation $G_1 \triangledown G_2$ of two grammars G_1 and G_2 has two properties:

1. $G_1 \leq G_1 \triangledown G_2$ and $G_2 \leq G_1 \triangledown G_2$.
2. For all increasing chains $G^{(0)}$, ..., $G^{(k)}$, $G^{(k+1)}$, ... in the abstract domain, the chain $G^{\triangledown(0)} = G^{(0)}$, ..., $G^{\triangledown(k+1)} = G^{\triangledown(k)} \triangledown G^{(k+1)}$, ... eventually stabilizes.

To compute $f_0^{i_0}(G_0)$, we compute $f_0^{i_0\,\triangledown(0)}(G_0) = AB$ and $f_0^{i_0\,\triangledown(k+1)}(G_0) = f_0^{i_0\,\triangledown(k)}(G_0) \triangledown f_{0\ \triangledown}^{i_0(k+1)}(G_0)$, where $f_{0\ \triangledown}^{i_0(k+1)}(G_0)$ is computed in an on-demand fashion by using the values of $f^{i\,\triangledown(k)}$'s rather than $f^{i(k)}$'s, until all $f^{i\,\triangledown}(G)$'s involved stabilize. This approximates the possibly non-existing fixed-point, forcing the iteration to terminate while guaranteeing that the sufficiency conditions are satisfied.

Widening operations have been defined implicitly [4, 38] or explicitly [12] for regular tree grammars. The idea is to enforce the use of deterministic regular tree grammars, *i.e.*, grammars that do not produce $N \stackrel{s}{\Rightarrow} c^n(N_1, ..., N_n)$ and $N \stackrel{s}{\Rightarrow} c^n(N_1', ..., N_n')$. For grammars with only productions of the form (5), this means that the grammars do not have two different productions of the form $N \Rightarrow c^n(N_1, ..., N_n)$ and $N \Rightarrow c^n(N_1', ..., N_n')$. Finding an appropriate widening operation is difficult. For example, we found that the so-called widening operation proposed by Cousot and Cousot [12] does not satisfy the second condition in the definition. We describe below an appropriate widening operation for regular-tree-grammar-based projections.

To facilitate widening, for every $f^i(G)$ used in the iteration, we associate a unique tag with its initial value AB. Also, after each iteration, we turn the resulting projection into an equivalent grammar whose production set contains only productions of the form (5).

The algorithm for computing $G_1 \triangledown G_2$ consists of repeatedly applying to $G = G_1 \vee G_2$ the following three steps:

1. Replace the productions $N \Rightarrow T_c(N_{i1}, ..., N_{in})$ for $i = 1..m$ where $m > 1$ with the production $N \Rightarrow T_c(N_1, ..., N_n)$ and the productions $N_j \Rightarrow N_{ij}$ for $i = 1..m$ and $j = 1..n$, where $N_1, ..., N_n$ are nonterminals not used in G.
2. For each N_{ij}, replace all its occurrences in all productions by N_j. If $N_j \Rightarrow AB_{tag}$ for an AB_{tag}, replaced all occurrences of the AB_{tag} in all productions by N_j.
3. Simplify the resulting grammar, *i.e.*, eliminate useless productions such as $M \Rightarrow M$ or $M_1 \Rightarrow M_2$ where M_1 is not reachable from the start symbol. Note that an AB may be simplified away, but an AB_{tag} is treated as a special terminal that can not be simplified away, *e.g.*, $N \Rightarrow AB_{tag} \mid T_{nil}$ can not be simplified to $N \Rightarrow T_{nil}$.

Each iteration makes one nonterminal N in G_1 and G_2 to be on the left hand side of one production of the form $N \Rightarrow T_c(N_1, ..., N_n)$. Therefore, at most $O(\mathcal{N}_1 + \mathcal{N}_2)$ iterations are needed. Note that this assumes that for each nonterminal N, there are a constant number of terminals T_c such that $N \Rightarrow T_c(N_1, ..., N_n)$. A more careful analysis is straightforward.

When the overall calculation terminates, simply take out the tag on any remaining tagged AB.

Example 6.2 We compute $odd^{1\,\nabla(k+1)}(ID) = odd^{1\,\nabla(k)}(ID) \mathbin{\nabla} odd1_{\nabla}^{(k+1)}(ID)$ and $even^{1\,\nabla(k+1)}(ID) = even^{1\,\nabla(k)}(ID) \mathbin{\nabla} even1_{\nabla}^{(k+1)}(ID)$ for the *odd* and *even* example. They stabilize after four iterations. The widenings for computing $odd^{1\,\nabla(k+1)}(ID)$ are given in detail; the widenings for computing $even^{1\,\nabla(k+1)}(ID)$ are similar.

	$odd^1{}_\nabla(ID)$	$even^1{}_\nabla(ID)$	$odd^{1\nabla}(ID)$	$even^{1\nabla}(ID)$
(0)			AB_o	AB_e
(1)	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(ID, AB_e)\}$	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(AB, AB_o)\}$	$\{S \Rightarrow AB_o \mid$ $T_{nil} \mid T_{cons}(ID, AB_e)\}$	$\{S \Rightarrow AB_e \mid$ $T_{nil} \mid T_{cons}(ID, AB_o)\}$
(2)	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(ID, N),$ $N \Rightarrow AB_e \mid$ $T_{nil} \mid$ $T_{cons}(AB, AB_o)\}$	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(AB, M),$ $M \Rightarrow AB_o \mid$ $T_{nil} \mid$ $T_{cons}(ID, AB_e)\}$	combine productions for S: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_1),$ $N_1 \Rightarrow AB_e \mid N,$ $N \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, AB_o)\}$ replace AB_e and N by N_1: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_1),$ $N_1 \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o)\}$	similarly as left: $\{S \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M_1),$ $M_1 \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$
(3)	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(ID, N_2),$ $N_2 \Rightarrow AB_e \mid$ $T_{nil} \mid$ $T_{cons}(AB, M),$ $M \Rightarrow T_{nil} \mid$ $T_{cons}(ID, AB_e)\}$	$\{S \Rightarrow T_{nil} \mid$ $T_{cons}(AB, M_2),$ $M_2 \Rightarrow AB_o \mid$ $T_{nil} \mid$ $T_{cons}(ID, N),$ $N \Rightarrow T_{nil} \mid$ $T_{cons}(AB, AB_o)\}$	combine productions for S: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_3),$ $N_3 \Rightarrow N_1 \mid N_2$ $N_1 \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o),$ $N_2 \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M),$ $M \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$ replace N_1 and N_2 by N_3: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_3),$ $N_3 \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o),$ $N_3 \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M),$ $M \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$ combine productions for N_3: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_3),$ $N_3 \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M_{11}),$ $M_{11} \Rightarrow AB_o \mid M,$ $M \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$ replace AB_o and M by M_{11}: $\{S \Rightarrow M_{11} \mid T_{nil} \mid T_{cons}(ID, N_3),$ $N_3 \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M_{11}),$ $M_{11} \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$ combine productions for S: $\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow N_3 \mid AB_e,$ $N_3 \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M_{11}),$ $M_{11} \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$ replace N_3 and AB_e by N_4: $\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow T_{nil} \mid T_{cons}(AB, M_{11}),$ $M_{11} \Rightarrow T_{nil} \mid T_{cons}(ID, N_4)\}$ simplify: $\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$	similarly as left: $\{S \Rightarrow T_{nil} \mid T_{cons}(AB, M_4),$ $M_4 \Rightarrow T_{nil} \mid T_{cons}(ID, S)\}$
(4)	...	...	$\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$	$\{S \Rightarrow T_{nil} \mid T_{cons}(AB, M_4),$ $M_4 \Rightarrow T_{nil} \mid T_{cons}(ID, S)\}$

This widening operation leads to iterations that always terminate, for two reasons. First, by Step 1, the number of productions for a nonterminal is finite, since the number of constructors in a given program is finite; also, each production has a finite right hand side. Second, by Step 2, the number of nonterminals is bounded, because more nonterminals are introduced only when the resulting projections grow as a result of recursive calls, but the replacements stop such growth by forcing the values of recursive calls to be approximated using existing nonterminals. To be precise, the potential sources of growth are tagged AB's in the resulting projections after each iteration, because each tagged AB comes from an application $f^i(G)$ whose approximations may form a strictly increasing chain. For any tagged AB's, say, $AB_{f^i(G)}$, if it is indeed a source of growth, then after a finite number of iterations, it occurs as an argument in a constructor application (otherwise, all tagged AB's do not occur as arguments of constructors after any finite number of iterations, we will reach a fixed point), and after another finite number of iterations, the position where $AB_{f^i(G)}$ occurs is replaced by something else, say Z (otherwise, the positions of all tagged AB's remain unchanged after any finite number of iterations, we will reach a fixed point). Then, in the widening after this iteration, we have, among other productions, $N \Rightarrow T_c(N_{11}, ...N_{1n})$, $N_{1j} \Rightarrow AB_{f^i(G)}$, $N \Rightarrow T_c(N_{21}, ...N_{2n})$, and $N_{2j} \Rightarrow Z$. By a use of Step 1, they will be replaced by $N \Rightarrow T_c(N_1, ...N_n)$, $N_j \Rightarrow N_{1j}$, $N_j \Rightarrow N_{2j}$, $N_{1j} \Rightarrow AB_{f^i(G)}$, and $N_{2j} \Rightarrow Z$; then, by a use of Step 2, all occurrences of $AB_{f^i(G)}$, as well as N_{1j}, will be replaced by N_j. Therefore, N_j approximates the result of $f^i(G)$ at all its occurrences, which stops the growth that comes from this result. Since there is a finite number of $f^i(G)$'s, *i.e.*, uniquely tagged AB's, in the worse case, they will all be replaced by some nonterminals, and we reach a fixed point.

In particular, if the definition of $f_0^{i_0}(G_0)$ forms recursive equations that involve only a finite set of $f_1^{i_1}(G_1)$'s, then the corresponding grammar G', formed from these recursive equations by associating with each $f_1^{i_1}(G_1)$ a unique nonterminal, is the least fixed point of the equations. For example, the definitions of $odd^1(ID)$ and $even^1(ID)$ form recursive equations:

$$\begin{aligned} odd^1(ID) &= T_{nil} \mid T_{cons}(ID, even^1(ID)) \\ even^1(ID) &= T_{nil} \mid T_{cons}(AB, odd^1(ID)) \end{aligned}$$

from which we obtain:

$$\begin{aligned} odd^1(ID) &= \{N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2), N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1)\} \\ even^1(ID) &= \{N_2 \Rightarrow T_{nil} \mid T_{cons}(AB, N_1), N_1 \Rightarrow T_{nil} \mid T_{cons}(ID, N_2)\} \end{aligned}$$

To prove that this claim is true, assume that G'' is the least fixed point. Notice that any sentence generated by G' must be generated by G''. Thus, by definition, $G' \leq G''$, *i.e.*, G' must be the least fixed point. Our widening operation gives precisely this least fixed point in such a case. However, this result allows us to obtain the least fixed point without the iterative computation.

Approximating the resulting projections within each iteration. To make the overall analysis results more precise, we developed another approximation operation $G_1 \forall G_2$ for two deterministic grammars G_1 and G_2. $G_1 \forall G_2$ is the most precise deterministic grammar above G_1 and G_2, *i.e.*, $G_1 \leq G_1 \forall G_2$ and $G_2 \leq G_1 \forall G_2$ and, for all deterministic G such that $G_1 \leq G$ and $G_2 \leq G$, $G_1 \forall G_2 \leq G$.

The algorithm for computing $G_1 \forall G_2$ consists of repeatedly applying to $G = G_1 \vee G_2$ the following three steps:

1. Replace the productions $N \Rightarrow Z$ for all Z with the productions $N \Rightarrow R$ for all R such that $N \overset{s}{\Rightarrow} R$. Replace the productions $N \Rightarrow T_c(N_{i1}, ..., N_{in})$ for $i = 1, 2$ with the production $N \Rightarrow T_c(N_1, ..., N_n)$ and the productions $N_j \Rightarrow N_{ij}$ for $i = 1, 2$ and $j = 1..n$, where $N_1, ..., N_n$ are nonterminals not used in G.

2. For each N_j, if there is a nonterminal N'_j not in G_1 or G_2 such that $N'_j \Rightarrow N_{ij}$ for $i = 1, 2$, then replace all occurrences of N_j by N'_j.
3. Simplify the resulting grammar, *i.e.*, eliminate useless productions such as $M \Rightarrow M$ or $M_1 \Rightarrow M_2$ where M_1 is not reachable from the start symbol. Note that AB's may be simplified away, *e.g.*, $N \Rightarrow AB \,|\, T_{nil}$ may be simplified to $N \Rightarrow T_{nil}$.

Each N_{ij} involved is a nonterminal in G_1 or G_2. Therefore, at most $O(\mathcal{N}_1\mathcal{N}_2)$ iterations are needed; similar to the analysis of widening, this assumes that for each nonterminal N, there are a constant number of terminals T_c such that $N \stackrel{s}{\Rightarrow} T_c(N_1, ..., N_n)$. Note that this is also the order of the size of the resulting grammar. So, this algorithm is optimal in this sense.

Example 6.3 Suppose grammar G_1 projects every second ($2n$th) element in a list, and grammar G_2 projects every third ($3n$th) element:

$$G_1 = \left\{ \begin{array}{ll} S_1 \Rightarrow T_{nil}, & A \Rightarrow T_{nil}, \\ S_1 \Rightarrow T_{cons}(AB, A), & A \Rightarrow T_{cons}(ID, S_1) \end{array} \right\}, G_2 = \left\{ \begin{array}{lll} S_2 \Rightarrow T_{nil}, & A_1 \Rightarrow T_{nil}, & A_2 \Rightarrow T_{nil}, \\ S_2 \Rightarrow T_{cons}(AB, A_1), & A_1 \Rightarrow T_{cons}(AB, A_2), & A_2 \Rightarrow T_{cons}(ID, S_2) \end{array} \right\}$$

G_1 never uses the $2n + 1$th elements, and G_2 never uses the $3n + 1$th and the $3n + 2$th elements. Combining them, we obtain $G = G_1 \curlyvee G_2$, which never uses the $6n+1$th and the $6n+5$th elements.

iteration	step	compute the least upper bound: $\begin{array}{l} S \Rightarrow S_1 \\ S \Rightarrow S_2 \end{array}$,where productions with S_1 and S_2 are as above.
1	1	combine $\begin{array}{l} S \Rightarrow S_1 \Rightarrow T_{nil}, \\ S \Rightarrow S_2 \Rightarrow T_{nil} \end{array}$ combine $\begin{array}{l} S \Rightarrow S_1 \Rightarrow T_{cons}(AB, A) \\ S \Rightarrow S_2 \Rightarrow T_{cons}(AB, A_1) \end{array}$: $\begin{array}{ll} S \Rightarrow T_{nil}, & B \Rightarrow A, \\ S \Rightarrow T_{cons}(AB, B), & B \Rightarrow A_1 \end{array}$
2	1	combine $\begin{array}{l} B \Rightarrow A \Rightarrow T_{nil} \\ B \Rightarrow A_1 \Rightarrow T_{nil} \end{array}$, combine $\begin{array}{l} B \Rightarrow A \Rightarrow T_{cons}(ID, S_1) \\ B \Rightarrow A_1 \Rightarrow T_{cons}(AB, A_2) \end{array}$: $\begin{array}{ll} B \Rightarrow T_{nil}, & C \Rightarrow S_1, \\ B \Rightarrow T_{cons}(ID, C), & C \Rightarrow A_2 \end{array}$
3	1	combine $\begin{array}{l} C \Rightarrow S_1 \Rightarrow T_{nil} \\ C \Rightarrow A_2 \Rightarrow T_{nil} \end{array}$, combine $\begin{array}{l} C \Rightarrow S_1 \Rightarrow T_{cons}(AB, A) \\ C \Rightarrow A_2 \Rightarrow T_{cons}(ID, S_2) \end{array}$: $\begin{array}{ll} C \Rightarrow T_{nil}, & D \Rightarrow A, \\ C \Rightarrow T_{cons}(ID, D), & D \Rightarrow S_2 \end{array}$
4	1	combine $\begin{array}{l} D \Rightarrow A \Rightarrow T_{nil} \\ D \Rightarrow S_2 \Rightarrow T_{nil} \end{array}$, combine $\begin{array}{l} D \Rightarrow A \Rightarrow T_{cons}(ID, S_1) \\ D \Rightarrow S_2 \Rightarrow T_{cons}(AB, A_1) \end{array}$: $\begin{array}{ll} D \Rightarrow T_{nil}, & E \Rightarrow S_1, \\ D \Rightarrow T_{cons}(ID, E), & E \Rightarrow A_1 \end{array}$
5	1	combine $\begin{array}{l} E \Rightarrow S_1 \Rightarrow T_{nil} \\ E \Rightarrow A_1 \Rightarrow T_{nil} \end{array}$, combine $\begin{array}{l} E \Rightarrow S_1 \Rightarrow T_{cons}(AB, A) \\ E \Rightarrow A_1 \Rightarrow T_{cons}(AB, A_2) \end{array}$: $\begin{array}{ll} E \Rightarrow T_{nil}, & F \Rightarrow A, \\ E \Rightarrow T_{cons}(AB, F), & E \Rightarrow A_2 \end{array}$
6	1	combine $\begin{array}{l} F \Rightarrow A \Rightarrow T_{nil} \\ F \Rightarrow A_2 \Rightarrow T_{nil} \end{array}$, combine $\begin{array}{l} F \Rightarrow A \Rightarrow T_{cons}(ID, S_1) \\ F \Rightarrow A_2 \Rightarrow T_{cons}(ID, S_2) \end{array}$: $\begin{array}{ll} F \Rightarrow T_{nil}, & G \Rightarrow S_1, \\ F \Rightarrow T_{cons}(ID, G), & G \Rightarrow S_2, \end{array}$
6	2	replace all occurrences of G by S, *i.e.*, replace $F \Rightarrow T_{cons}(ID, G)$ by $F \Rightarrow T_{cons}(ID, S)$.

Simplifications at each step are straightforward and thus are not shown explicitly. We obtain:

$$G = \left\{ \begin{array}{llllll} S \Rightarrow T_{nil}, & B \Rightarrow T_{nil}, & C \Rightarrow T_{nil}, & D \Rightarrow T_{nil}, & E \Rightarrow T_{nil}, & F \Rightarrow T_{nil}, \\ S \Rightarrow T_{cons}(AB, B), & B \Rightarrow T_{cons}(ID, C), & C \Rightarrow T_{cons}(ID, D), & D \Rightarrow T_{cons}(ID, E), & E \Rightarrow T_{cons}(AB, F), & F \Rightarrow T_{cons}(ID, S) \end{array} \right\}$$

We can now use $G_1 \curlyvee G_2$ in place of $G_1 \vee G_2$ in our analysis. Since $G_1 \vee G_2 \leq G_1 \curlyvee G_2$, the sufficiency conditions still hold. Moreover, even though $G_1 \curlyvee G_2$ may give a less precise grammar than $G_1 \vee G_2$ when used within one iteration by making $G_1 \vee G_2$ deterministic, when combined with the widening operation after each iteration, which gives a much less precise deterministic grammar, the overall analysis result is more precise than not using the $G_1 \curlyvee G_2$ operation.

These approximation operations are designed for fully general regular-tree-grammar-based projections. Applying them allows us to approximate the possibly non-existing fixed-points and at the same time obtain quite precise analysis results. The resulting grammars are not from any fixed abstract domain; they follow from the structures of programs and their sizes are bounded by certain functions of program sizes.

7 Analyzing and eliminating dead computations

Given a projection G_0 that describes how much of the result of a function f_0 is live, we can compute for each subexpression in the function definitions an associated projection describing how much of the value of the subexpression is live and then eliminate dead subexpressions. This is easy based on the analysis in Section 4.

First, the definitions are similar. The projection that is applied to the result of a function is the projection that is applied to its defining expression. The projection applied to each subexpression can be computed from the projection applied to its super expression. For example, if the projection applied to an expression $c(e_1, ..., e_n)$ is G, then the projection applied to its subexpression e_i is $G_{c_{\bar{i}}}$; if the projection applied to an expression $f(e_1, ..., e_n)$ is G, then the projection applied to the result of f is G, and the projection applied to the subexpression e_i is $f^i(G)$, which can be computed as in Section 4.

Then, we compute all the projections by starting at G_0 for the result of f_0 and AB for all other indices and iterating using the above definitions. The same methods can be used to guarantee termination.

Finally, we eliminate all dead computations in a program by replacing all subexpressions whose associated projections are AB with $\_$. For example, for the function $\bar{b}'$ in Example 5.2, we replace the third component in the second and third branch with $\_$. We can save more space by replacing constructors to remove dead components completely. For example, we can replace *triple* by *pair*, with overloaded selectors $1st$ and $2nd$, and obtain:

$$
\begin{array}{l}
\bar{b}'(i,j,\bar{r}) \\
= \ \textbf{if}\ j=0\ \vee j=i+1\ \textbf{then}\ pair(1,\_) \\
\quad\ \textbf{else if}\ j=i\ \textbf{then} \\
\quad\quad\quad \textbf{let}\ v=\bar{b}'(i-1,j-1,pair(1,\_)) \\
\quad\quad\quad \textbf{in}\ \ pair(1st(v)+1,v) \\
\quad\ \textbf{else let}\ v=\bar{b}'(i-1,j-1,2nd(\bar{r})) \\
\quad\quad\quad \textbf{in}\ \ pair(1st(v)+1st(\bar{r}),v)
\end{array}
$$

Our transformation preserves semantics in the sense that, if the original program terminates with a value that is not an error, then the new program terminates with the same value.

If the original program does not terminate, or terminates with an error, such as dividing by zero or selecting from a wrong construction, the new program may or may not behave the same. The differences are (i) if the original program does not terminate, then the new program may terminate with a value or with an error and (ii) if the original program terminates with an error, the new program may terminate with a value or a different error. We can make our analysis more careful so that the new program behaves the same as the original one on more inputs that cause error. Reps and Turnidge do this partially by doing a separate shape analysis [36], but they do not report any error after all. In any case, errors can be handled only partially due to undecidabilities, thus we can not make the optimized program behave exactly the same as the original program. This difference is unavoidable for our language that has call-by-value semantics. We can eliminate such difference for a similar language with appropriate call-by-name semantics. We choose call-by-value because it is more widely used.

8 Applications and discussions

Our analyses and transformations help understand and optimize programs, as well as modify and reuse programs.

Typed languages. We describe our methods for an untyped language, but they apply to typed languages as well. For a typed language, possible values are restricted also by type information so that the overall analysis results can be more precise, *e.g.*, type information about the value of an expression e can help restrict the set CON in computing $e^v(CON)$ for some variable v. In a typed language, we can also obtain a finite grammar abstract domain directly from the data type information and use it to guarantee the termination of the analysis.

Typing. Our analysis can be viewed as automatic inference of a special kind of type information that helps understand and check programs. For computing the result of a program, if a projection G is associated with a variable, then that much of the data is possibly needed. In any particular run, less than G could actually be needed, but otherwise the data must be consistent with G. This type information also indicates dead data parts whose types are not of interest. In the presence of recursive data types, it determines partially dead recursive types.

Slicing. Starting at a particular index in the program, not necessarily the final result of the entire program, the analysis helps slice out data and computations that are possibly needed for that index. This is called backward slicing [42], and it helps debug and reuse program pieces. It is essentially dead code analysis used for other software engineering purposes. It can also be regarded as a kind of program specialization with respect to program output [36].

Dead code elimination. This is a most traditional compiler optimization [3] and is a most straightforward application of our analysis and transformation. Since programmers do not intentionally write much dead code, this optimization is usually most effective when combined with other code optimizations or reorganizations, such as deforestation, incrementalization, and code reuse.

Deforestation and *fusion.* These optimizations combine function applications to avoid building large intermediate results, *i.e.*, results of function calls to be passed as arguments to other function calls [40, 7]. To guarantee that the optimizations can be done effectively, the functions and subexpressions involved must satisfy certain conditions, *e.g.*, be in blazed treeless forms [40]. Our analysis can help identify dead functions and subexpressions not satisfying these conditions and prune them out, thus making deforestation more widely applicable.

Incrementalization, *finite differencing*, and *strength reduction.* These optimizations focus on finding and replacing subcomputations whose values can be retrieved from the result of a previous computation, and they can often achieve asymptotic speedup [8, 32, 28, 26]. Dead code elimination is the last step in such optimizations; it removes computations whose values were used in computing the replaced subcomputations and is crucial for the overall speedup.

Caching intermediate results for program improvement. This is related to incrementalization, as has been discussed in Section 4. However, the cache-and-prune method [27] allows us to achieve drastic program optimizations that are not possible otherwise. We have used this method in deriving a collection of dynamic programming programs found in standard texts [2, 34, 10]. Powerful and automatic pruning techniques are essential in such optimizations.

The methods in this paper are being implemented in the Synthesizer Generator [35]. The implementation involves efficient algorithms on regular tree grammars and their uses in program analysis [4, 29].

9 Related work and conclusion

Our backward dependence analysis uses domain projections to specify sufficient information. Wadler and Hughes use projections for strictness analysis [41]. 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 [24]. It is a forward analysis and is proved equivalent to strictness analysis [25]. Mogensen also uses projections, based on grammars in particular, to do binding-time analysis, but he uses only a restricted class of regular tree grammars [31].

Several analyses are in the same spirit as ours, even though some do not use the name projection. The necessity interpretation by Jones and Le Métayer [21] uses necessity patterns that correspond to projections. Necessity patterns specify only heads and tails of list values. The absence analysis by Hughes [18] uses the name context in place of projection. 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 and Teitelbaum [27] uses projections to specify specific components of tuple values and thus provide more accurate information. However, methods used there for handling unbounded growth of such projections are crude. The analysis in this paper represents projections using regular tree grammars extended with live and dead, which more precisely describe partially dead data structures and more naturally guarantee bounded growth.

The idea of using regular tree grammars for program flow analysis is due to Jones and Muchnick [19], 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 [30, 31, 5, 38, 36]. In particular, the analysis for backward slicing by Reps and Turnidge [36] 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 hand side of one production, and each right hand side is one of five forms, corresponding to *ID*, *AB*, atom, pair, and atom | pair. Our work uses general regular tree grammars extended with *ID* and *AB*. We also use additional production forms, such as selector form, to make the framework more flexible. Compared with that work, we also handle more program constructs, namely, binding expressions and user-defined constructors of arbitrary arity. To summarize, our analysis applies to general recursive data constructions and produces more precise analysis results than previous methods.

We believe that our treatment is also more rigorous, since we adopt the view that regular-tree-grammar-based program analysis is also abstract interpretation [12]. We extend the grammars and handle *ID* and *AB* specially in grammar ordering, equivalence, and approximation operations. Our transformers for dead code analysis are general. Our methods for computing them terminate by either using appropriate finite grammar domains or combining carefully designed approximation operations. In particular, we describe how to use finite grammar domains for pruning analysis in a new application [27], and we develop a previously unknown approximation operation that allows us to compute very precise analysis result. Such operations are difficult to design, *e.g.*, we found that the so-called widening operation given in a previous work [12] may still lead to an infinitely increasing chain. While regular-tree-grammar-based program analysis can be reformulated as set-constraint-based analysis [16, 17, 12], we do not know any work that treats precise dead code analysis as we do.

To conclude, the overall goal is to analyze dead data and eliminate computations on them across recursions and loops, possibly interleaved with wrappers like classes in object oriented programming styles. This paper discusses techniques for recursion. The basic ideas should extend to loops. A recent work has just started this direction; it extends slicing to symbolically capture particular iterations in a loop [33]. Object-oriented programming style is used widely, but cross-class optimization heavily depends on inlining, which often causes code blow-up. Grammar-based analysis and transformation used for cross-class optimization without inlining.

Acknowledgments

The author would like to thank Byron Long for taking the effort to implement the methods in this paper, Scott Stoller for a number of suggestions for improving the methods and the paper, and both of them for many helpful discussions. The author is also grateful to Alex Aiken, Chris Colby, and Tom Reps for discussions about related work.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Series in Computers and Their Applications. E. Horwood, Chichester; Halsted Press, New York, 1987.

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

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

[4] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Proceedings of the 5th ACM Conference on FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 427–447, Cambridge, Massachusetts, August 1991. Springer-Verlag, Berlin.

[5] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the 18th Annual ACM Symposium on POPL*, January 1991.

[6] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN '97 Conference on PLDI*, pages 159–170, Las Vegas, Nevada, June 1997.

[7] W.-N. Chin. Safe fusion of functional expressions. In *Proceedings of the 1992 ACM Conference on LFP*, pages 11–20, June 1992.

[8] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[9] J. Cocke and J. T. Schwartz. Programming Languages and Their Compilers; Preliminary Notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 14th Annual ACM Symposium on POPL*, pages 238–252, Los Angeles, California, Jan. 1977.

[12] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 17th International Conference on FPCA*, pages 170–181, La Jolla, California, June 1995.

[13] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

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

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

[17] N. Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LFP*, pages 306–317, June 1994.

[18] J. Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, chapter 5, pages 117–153. Addison-Wesley, Reading, Massachusetts, 1999.

[19] 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 POPL*, pages 244–256, San Antonio, Texas, January 1979.

[20] 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*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[21] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th International Conference on FPCA*, pages 54–74, London, U.K., September 1989.

[22] K. Kennedy. Use-definition chains with applications. *Journal of Computer Languages*, 3(3):163–179, 1978.

[23] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on PLDI*, pages 147–158, Orlando, Florida, June 1994.

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

[25] J. Launchbury. Strictness and binding-time analysis: Two for the price of one. In *Proceedings of the ACM SIGPLAN '91 Conference on PLDI*, pages 80–91, Toronto, Ontario, Canada, June 1991.

[26] 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 POPL*, pages 157–170, St. Petersburg Beach, Florida, January 1996.

[27] Y. A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, pages 190–201, La Jolla, California, June 1995.

[28] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, February 1995.

[29] D. Melski and T. Reps. Interconvertibility of set constraints and conext-free language reachability. In *Proceedings of the ACM SIGPLAN Symposium on PEPM*, Amsterdam, The Netherlands, June 1997.

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

[31] T. Mogensen. Separating binding times in language specifications. In *Proceedings of the 4th International Conference on FPCA*, pages 12–25, London, U.K., September 1989.

[32] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[33] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. Technical Report CS-TR-3737, Department of Computer Science, University of Maryland, College Park, Maryland, April 1997.

[34] P. W. Purdom, Jr. and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, 1985.

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

[36] 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.

[37] 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. Lecture notes of 1981 Marktoberdorf Summer School on Theoretical Foundations of Programming Methodology, directed by F.L. Bauer, E.W. Dijkstra, and C.A.R. Hoare.

[38] 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, Edinburgh, U.K., April 1994. Springer-Verlag, Berlin.

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

[40] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. Special issue of selected papers from the 2nd ESOP.

[41] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407, Portland, Oregon, September 1987.

[42] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.