Dependence Analysis for Recursive Data

Yanhong A. Liu*

Abstract

This paper describes a general and powerful method for dependence analysis in the presence of recursive data constructions. The particular analysis presented is for identifying partially dead recursive data, but the general framework for representing and manipulating recursive substructures applies to all dependence analyses. The method uses projections based on general regular tree grammars extended with notions of live and dead, and defines the analysis as mutually recursive grammar transformers. To guarantee that the analysis terminates, we use carefully designed approximations. 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 to give the most precise deterministic result possible. All grammar operations used in the analysis have efficient algorithms. The overall analysis yields significantly more precise results than other known methods.

1 Introduction

Dependence analysis is the basis of compiler optimizations and program manipulation. Examples include liveness analysis for dead-code elimination [3], flow analysis for storage allocation [38], binding-time analysis for partial evaluation [25], and strictness analysis for optimizing lazy evaluation [1]. In particular, dead code produces values that never get used. Such code appears often as a result of program optimization, modification, and reuse [38, 3]. There are also other programming activities that do not explicitly involve live or dead code but rely on similar notions. Examples are program slicing [54, 46], program specialization [46], and compile-time garbage collection [27, 41]. Analysis for identifying live or dead code, or code of similar relevance, has been studied and used widely [10, 9, 28, 39, 3, 27, 23, 15, 29, 34, 33, 50, 46]. 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, dependence analyses have been made more precise so as to be effective in more complicated computations [53, 30, 37, 16, 41], and this is particularly true for dead-code analysis [23, 15, 29, 33, 46, 7]. Since recursive data constructions are used increasingly widely in high-level languages, an important problem is dependence analysis for recursive data. The goal of dead-code analysis here is to identify *partially dead recursive data*, *i.e.*, recursive data whose dead parts form recursive substructures. It is difficult because the structures of general recursive data can be user-defined, and dead recursive substructures are interleaved with other recursive substructures on which the computations are live. Several methods have been studied, but all have limitations [27, 23, 33, 46]. Similar limitations exist also in other analyses for recursive data [53, 30, 37, 16, 41].

This paper describes a general and powerful method for analyzing dependencies for recursive data. We represent partially dead recursive data using projections based on general regular tree grammars extended with notions 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 carefully designed approximations. 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 to give the most precise deterministic result possible. All grammar operations used in the analysis have efficient algorithms. The overall analysis yields consistently more precise results than other known methods.

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. Section 4 defines the analysis as recursive grammar transformers and proves its correctness. Sections 5 presents three approximation operations that together make the analysis precise and efficient. Section 6 discusses applications, extensions, and efficient algorithms for implementations. Section 7 compares with related work and concludes.

2 Language

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

$e ::= v$		variable
	$c(e_1, ..., e_n)$	constructor application
	$p(e_1, ..., e_n)$	primitive function application
	$f(e_1, ..., e_n)$	function application
	if e_1 **then** e_2 **else** e_3	conditional expression
	let $v = e_1$ **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 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; if $n > 0$, there is a primitive function

*This work is supported in part by NSF Grant CCR-9711253.
Author's address: Computer Science Department, Indiana University, Bloomington, IN 47405. Email: liu@cs.indiana.edu.

c_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 constructors nil^0 and $cons^2$ are declared. For ease of reading, we write *null* for $nil?$, *car* for $cons_1^-$, and *cdr* for $cons_2^-$ in programs.

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

$odd(x) =$ **if** $null(x)$ **then** nil
else $cons(car(x), even(cdr(x)))$

$even(x) =$ **if** $null(x)$ **then** nil
else $odd(cdr(x))$

$cut(n, l)$: 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

$cut(n, l) =$ **if** $n = 0$ **then** $cons(0, cdr(l))$
else $cons(n, cdr(cdr(cut(n - 1, cdr(l)))))$

Figure 1: Example function definitions

This language can have either call-by-value or call-by-name semantics. Well-defined expressions evaluate to constructed data, such as $cons(3, nil)$. We use $\perp$ to denote the value of undefined (non-terminating) expressions. Since a program can use data constructions $c(e_1, ..., e_n)$ in recursive function definitions, it can build data structures of unbounded size.

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 (or the condition, respectively) evaluates to $\_$. Whether a function application (or a binding expression) evaluates to $\_$ depends on the values of the arguments (or the bound variable, respectively) and how they are used in the function definition (or the body, respectively). If the language is call-by-value, then the resulting program terminates with the correct value whenever the original program does; if the language is call-by-name, then the resulting program terminates with the correct value exactly when the original program does.

3 Representing partially dead recursive data

Projections. Domain projections [47, 20] can be used to project out parts of data that are of interest [53, 30, 37, 46], in our case, the live parts. Let X be the domain of all possible values computed by our programs, including $\perp$ and values containing $\_$. For all values x in X, $\perp \sqsubseteq x$; for two values x_1 and x_2 other than $\perp$,

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

A *projection* is a function $\pi : X \to X$ such that $\pi(x) \sqsubseteq x$ and $\pi(\pi(x)) = \pi(x)$ for all $x \in X$. Two special projections are *ID* and *AB*. *ID* is the identity function: $ID(x) = x$. *AB* is the absence function: $AB(x) = \_$ for all $x \neq \perp$, 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:

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

If a constructor c has arity 0, then we use T_c in place of $T_c()$. For example,

$$T_{cons}(AB, T_{cons}(ID, AB))(cons(0, cons(1, cons(2, nil)))) = cons(\_, cons(1, \_)).$$

To summarize, a projection is a function; when applied to constructed data, it returns the data with the dead parts, *i.e.*, parts corresponding to *AB*, replaced by $\_$.

Grammar-based projections. Regular tree grammars have been used to describe recursive substructures and other data flow information [26, 36, 37, 5, 48, 14, 46]. We describe partially dead recursive data using projections that are represented using regular tree grammars. A *regular-tree-grammar-based projection* G, called regular tree grammars for short, is a quadruple $\langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$, where $\mathcal{T}$ is a set of terminal symbols including *ID*, *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 forms:

$$N \Rightarrow ID, \quad N \Rightarrow AB, \quad \text{or} \quad N \Rightarrow T_{c^n}(N_1, ..., N_n), \tag{4}$$

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

$$G(x) = \sqcup\{\pi(x) \mid \pi \in L_G\}. \tag{5}$$

where $\sqcup$ is the least upper bound operation for $\sqsubseteq$. It's easy to see that $G(x)$ is well-defined for $x \in X$. We overload *ID* to denote the grammar that contains a single production $S \Rightarrow ID$, and overload *AB* to denote the grammar that contains a single production $S \Rightarrow AB$. For ease of presentation, when no confusion arises, we use the start symbol, with associated productions when necessary, possibly in compact forms, to denote a grammar-based projection. For example, $\{S \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$, which is a grammar with one production in compact form, projects out a list with only its spine but not any elements.

For convenience, we extend regular tree grammars to allow productions of the forms:

$$N \Rightarrow N' \quad \text{or} \quad N \Rightarrow T_{c_i^-}(N'), \tag{6}$$

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

$$T_{c_i^-}(\pi) = \begin{cases} ID & \text{if } \pi = ID \\ \pi_i & \text{if } \pi = T_{c^n}(\pi_1, ..., \pi_n) \\ AB & \text{otherwise} \end{cases} \tag{7}$$

Given such an extended regular tree grammar G that contains productions of the forms in (4) and (6), we can define a relation $\stackrel{s}{\Rightarrow}$ to shortcut productions of the forms in (6):

$$\begin{array}{l} N \stackrel{s}{\Rightarrow} R \text{ iff } N \Rightarrow R \text{ is of a form in (4), or} \\ \qquad N \Rightarrow N' \text{ and } N' \stackrel{s}{\Rightarrow} R, \text{ or} \\ \qquad N \Rightarrow T_{c_i^-}(N'),\ N' \stackrel{s}{\Rightarrow} T_{c^n}(N_1,...,N_n),\ \text{and } N_i \stackrel{s}{\Rightarrow} R. \end{array} \quad (8)$$

Then, we can construct a regular tree grammar G' that contains only productions of the forms in (4) such that $L_G = L_{G'}$. The construction and proof are similar to when only the selector form is used [26]. Hereafter, 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 that index is live.

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

$$G_1 \leq G_2 \quad \text{iff} \quad \forall \pi_1 \in L_{G_1},\ \exists \pi_2 \in L_{G_2},\ \pi_1 \leq \pi_2, \quad (9)$$

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

$$\begin{array}{l} \pi_1 \leq \pi_2 \quad \text{iff} \quad \pi_1 = \mathit{AB},\ \text{or}\ \ \pi_2 = \mathit{ID},\ \text{or} \\ \qquad \pi_1 = T_c(\pi_{11},...,\pi_{1n}),\ \pi_2 = T_c(\pi_{21},...,\pi_{2n}), \\ \qquad\qquad \text{and } \pi_{1i} \leq \pi_{2i} \text{ for } i = 1..n. \end{array} \quad (10)$$

This ordering is decidable because, given G_1 and G_2, we can construct G_1' and G_2' such that $G_1 \leq G_2$ iff $L_{G_1'} \subseteq L_{G_2'}$, and the latter is decidable [19, 4]. The construction of G_i' (for $i = 1, 2$) from G_1 and G_2 has three steps; assuming nonterminals are renamed so that those in G_1 are distinct from those in G_2:

1. Let $G_i' = G_i$.
2. If $N \Rightarrow \mathit{ID} \in \mathcal{P}_i$, then add to $\mathcal{P}_i'$ the production $N \Rightarrow \mathit{AB}$ 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 $N \Rightarrow \mathit{AB}$.

It is easy to see that

$$\text{if } G_1 \leq G_2,\ \text{then } \forall x (G_1(x) \sqsubseteq G_2(x)). \quad (11)$$

The converse is not true, *e.g.*,

$$\begin{array}{l} G_1 = \{S \Rightarrow T_{\mathit{cons}}(\mathit{ID}, \mathit{ID})\}, \text{ and} \\ G_2 = \{S \Rightarrow T_{\mathit{cons}}(\mathit{ID}, \mathit{AB}) \mid T_{\mathit{cons}}(\mathit{AB}, \mathit{ID})\} \end{array}$$

form a counterexample.

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 nonterminals 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:

$$G_1 \vee G_2 = \langle \mathcal{T}_1 \cup \mathcal{T}_2,\ \mathcal{N}_1 \cup \mathcal{N}_2 \cup \{S\},\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S \Rightarrow S_1, S \Rightarrow S_2\},\ S \rangle \quad (12)$$

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.

The ordering (9) can be extended in a pointwise fashion (with respect to elements of $\mathcal{I}$) to the domain $\mathcal{D}$. 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 $N \Rightarrow \mathit{AB}$. Given a grammar $G = \langle \mathcal{T}, \mathcal{N}, \mathcal{P}, S \rangle$, we use $G_{c_i^-}$ 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:

$$G_{c_i^-} = \langle \mathcal{T},\ \mathcal{N} \cup \{S_i\},\ \mathcal{P} \cup \{S_i \Rightarrow T_{c_i^-}(S)\},\ S_i \rangle \quad (13)$$

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

$$\begin{array}{l} G_{c_i} = \langle \mathcal{T},\ \mathcal{N} \cup \{S_0\}, \\ \qquad \mathcal{P} \cup \{S_0 \Rightarrow T_{c^n}(\overbrace{N,...,N}^{i-1}, S, \overbrace{N,...,N}^{n-i}),\ N \Rightarrow \mathit{AB}\},\ S_0 \rangle \end{array} \quad (14)$$

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

$$\mathit{CON} = \{S \Rightarrow T_{\mathit{nil}} \mid T_{\mathit{cons}}(\mathit{AB}, \mathit{AB})\}.$$

If $G = \mathit{ID}$, then

$$\begin{array}{l} G_{\mathit{cons}_1^-} = G_{\mathit{cons}_2^-} = \mathit{ID} \\ G_{\mathit{cons}_1} = \{S \Rightarrow T_{\mathit{cons}}(\mathit{ID}, \mathit{AB})\} \\ G_{\mathit{cons}_2} = \{S \Rightarrow T_{\mathit{cons}}(\mathit{AB}, \mathit{ID})\}. \end{array}$$

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. The analysis is based on *projection transformers*.

Transformers for grammar-based projections. We use f^i to denote a *projection transformer* that takes a projection associated with the result of f and returns a projection associated with the ith parameter; f^i must satisfy the *sufficiency condition*: if $G_i = f^i(G)$, then

$$G(f(v_1, ..., v_i, ..., v_n)) \sqsubseteq f(v_1, ..., G_i(v_i), ..., v_n) \quad (15)$$

for all values of $v_1, ..., v_n$. Similarly, we use e^v to denote a projection transformer that takes a projection associated with e and returns a projection associated with every instance of v in e; a similar sufficiency condition must be satisfied: if $G' = e^v(G)$, then

$$G(e) \sqsubseteq e[G'(v)/v] \quad (16)$$

for all values of the variables in e.

For a function definition $f(v_1, ..., v_n) = e$, 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:

$$e^v(AB) = AB. \tag{17}$$

For $G \neq AB$, we define $e^v(G)$ in Figure 2. Rules (3)-(5) handle data constructions; as a special case of (3), for a constructor c of arity 0, the right hand side is AB. Rule (6) applies to all primitive functions other than selectors and testers. Rule (7) is the recursive definition using f^i's. Rule (8) follows from the semantics of conditionals. Rule (9) follows from the semantics of bindings, assuming $u \neq v$ after renaming. We

(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) $(q(e_1, ..., e_n))^v(G)$	$= {e_1}^v(ID) \vee ... \vee {e_n}^v(ID)$
(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) $(\textbf{let } u = e_1 \textbf{ in } e_2)^v(G)$	$= {e_1}^v({e_2}^u(G)) \vee {e_2}^v(G)$

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

can easily show by induction that each transformer is non-decreasing.

Theorem 4.1 *Transformers f^i and e^v satisfy the sufficiency conditions (15) and (16), respectively.*

Proof: Each rule guarantees sufficient information, and thus the sufficiency conditions are satisfied by induction. As an example, we show this for rule (3). First, we show that

$$G(c(e_1, ..., e_n)) \sqsubseteq c(G_{c_1^-}(e_1), ..., G_{c_n^-}(e_n)). \tag{18}$$

By definition of G in (5), we only need to show $\forall \pi \in L_G$, $\pi(c(e_1, ..., e_n)) \sqsubseteq c(G_{c_1^-}(e_1), ..., G_{c_n^-}(e_n))$. We show this in each of the three possible cases below. Let $\pi \in L_G$.

1. If $\pi = ID$, then by definitions of $G_{c_i^-}$ in (13) and $T_{c_i^-}$ in (7), we have $ID \in L_{G_{c_i^-}}$. Then, $\pi(c(e_1, ..., e_n)) = c(e_1, ..., e_n) = c(G_{c_1^-}(e_1), ..., G_{c_n^-}(e_n))$.
2. If $\pi = T_c(\pi_1, ..., \pi_n)$, then by definitions of $G_{c_i^-}$ in (13) and $T_{c_i^-}$ in (7), we have $\pi_i \in L_{G_{c_i^-}}$. Then, $\pi(c(e_1,...,e_n)) \overset{\text{by (3)}}{=} c(\pi_1(e_1),...,\pi_n(e_n)) \sqsubseteq c(G_{c_1^-}(e_1),...,G_{c_n^-}(e_n))$.
3. For other π, $\pi(c(e_1,...,e_n)) \overset{\text{by (3)}}{=} \bot \sqsubseteq c(G_{c_1^-}(e_1),...,G_{c_n^-}(e_n))$.

Now, we show the induction for the case of rule (3), *i.e.*, if (i) $G'_i = {e_i}^v(G_{c_i^-})$ and (ii) $G' = \bigvee_{i=1}^n G'_i$, then $G(c(e_1, ..., e_n)) \sqsubseteq c(e_1, ..., e_n)[G'(v)/v]$, as required by (16). We have

$$G_{c_i^-}(e_i) \sqsubseteq e_i[G'_i(v)/v] \sqsubseteq e_i[G'(v)/v], \tag{19}$$

where the first ordering is by (i) and induction hypothesis, and the second ordering is by (ii) and implication (11). Combining (18) and (19) yields $G(c(e_1, ..., e_n)) \sqsubseteq c(e_1, ..., e_n)[G'(v)/v]$. □

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

$$odd^1(G) = CON \vee (G_{cons_1^-})_{cons_1} \vee (even^1(G_{cons_2^-}))_{cons_2}$$
$$even^1(G) = CON \vee (odd^1(G))_{cons_2}$$
$$cut^2(G) = (G_{cons_2^-})_{cons_2} \vee (cut^2(((G_{cons_2^-})_{cons_2})_{cons_2}))_{cons_2}$$

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 $f_1^{i_1(0)}(G_2)$ 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 $\{S \Rightarrow T_{cons}(AB, ID)\}$. In the table below, each column contains successive approximations to the result of applying cut^2 to the projection at the top of the column.

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	AB	AB	...
(1)	$\{S \Rightarrow T_{cons}(AB, ID)\}$	$\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, ID))\}$	...	
(2)	$\{S \Rightarrow T_{cons}(AB, ID)\}$	...		
...	...			

Also, 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; the result does not even stabilize:

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

approximations of $odd^1(ID)$ and $even^1(ID)$ are strictly increasing:

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

There are three standard ways to guarantee termination of the iteration: using finite abstract domains, using finite transformers, or using approximation operations. 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 have studied this method and applied it to caching intermediate results for program improvement [33]. Finite transformers can be obtained by restricting them to be written in a specific meta-language [14]. This meta-language corresponds to a restricted class of regular tree grammars extended with selectors [26, 14] and can be rewritten as a set of constraints [21, 22, 14]. This is essentially a masking of the explicit use of an approximation operation, called widening, when defining transformers [14]. Approximation operations provide a more general solution and make the analysis framework more modular and flexible. We describe three efficient approximation operations that together allow our analysis to give more precise results than previous methods.

5 Approximation operations

Approximating argument projections. The goal is to guarantee that computing a transformer application $f^i(G)$ does not make f^i run into infinitely many new arguments. We compute $f_0^{i_0}(G_0)$ in an on-demand fashion, *i.e.*, 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, and 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 the latter. We use two data structures:

1. For each transformer f^i, we keep a list of all arguments on which it has been called.
2. For each k, we keep a stack of all transformer applications needed in computing $f_0^{i_0(k)}(G_0)$.

If, while some $f_1^{i_1(k_1)}(G_1)$ is in the stack, we need to recursively call $f_1^{i_1(0)}(G_2)$ for a new argument G_2 to $f_1^{i_1}$, then we will make sure that evaluating $f_1^{i_1(k_1)}(G_1)$ does not make $f_1^{i_1}$ run into infinitely many new arguments. As a safe case, if $size(G_2) < size(G_1)$ for a given well-founded measure $size$, then we continue normal on-demand evaluation. Each regular tree grammar corresponds to a finite tree automata, which can be minimized [19], so we can use the number of states as the measure.

If $size(G_2) \not< size(G_1)$, then we obtain, symbolically, from G_1 and the definitions of the transformers currently on the stack starting from $f_1^{i_1(k_1)}(G_1)$, a grammar G_1' that is a sufficient approximation of G_1, G_2, and possible future arguments following this cyclic path of recursive calls in the call graph, and we use $f_1^{i_1(k)}(G_1')$ in place of $f_1^{i_1(k)}(G_1)$ for all k. We construct G_1' as follows.

1. Follow the calls from $f_1^{i_1(k_1)}(G_1)$ to $f_1^{i_1(0)}(G_2)$ on the stack. List the definitions of all f^i's called, each definition being of the form $f^i(G) = ...f'^{i'}(G')...$, where calls to f^i and $f'^{i'}$ are consecutive on the stack; the first definition in the list is that of $f_1^{i_1}$, and the last definition contains the corresponding recursive call to $f_1^{i_1}$.
2. Let S denote the start symbol of G in the first definition $f_1^{i_1}(G) = ...$, and let S' denote the start symbol of G' in the last definition $... = ...f_1^{i_1}(G')....$ Represent S' symbolically in terms of S by following syntactically the definitions of the f^i's and obtain $\{S' \Rightarrow ..., \ ..., \ ... \Rightarrow ...S...\}$.
3. Let S_1 denote the start symbol of G_1, and define $G_1' = \{S \Rightarrow S_1, \ S \Rightarrow S'\}$, where the productions for S_1 are as in G_1 and the productions for S' are as obtained from Step 2.

This construction takes at most linear time in the size of the program since it follows the function definitions syntactically and visits each piece of code at most once.

We have $G_1 \leq G_1'$. Since the transformers are non-decreasing, we have $f_1^{i_1(k)}(G_1) \leq f_1^{i_1(k)}(G_1')$ for all k. Therefore, using G_1' is a conservative approximation of using G_1, so the sufficiency conditions still hold. Also, in computing $f_1^{i_1(k_1)}(G_1')$, the recursive call corresponding to $f_1^{i_1(0)}(G_2)$ will have the same argument G_1', and thus there will not be growth of argument projections following this cyclic path in the call graph. In particular, if G_1, G_2, and possible future arguments form an increasing chain, then G_1' is the least upper bound; if G_1, G_2, and possible future arguments form a decreasing chain, then G_1' is a conservative approximation that equals G_1 but is greater than the rest. Since, in the call graph, there are a finite number of simple cycles starting at any transformer (regardless of the argument), the above approximation prevents the evaluation of $f_1^{i_1(k_1)}(G_1)$ from making $f_1^{i_1}$ run into infinitely many new arguments.

Example 5.1 Consider the transformer cut^2 in Example 4.1. Let S be the start symbol for G, and S' be the start symbol for $G' = ((G_{cons_2^-})_{cons_2})_{cons_2}$, then

$$S' \Rightarrow T_{cons}(AB, M), \ M \Rightarrow T_{cons}(AB, N), \ N \Rightarrow T_{cons_2^-}(S).$$

In computing $cut^2(ID)$, the stack contains $cut^{2(1)}(ID)$ and $cut^{2(0)}(\{S \Rightarrow T_{cons}(AB, T_{cons}(AB, ID))\})$. We have $G_1 = ID$; we obtain $G_1' = \{S \Rightarrow ID, \ S \Rightarrow S'\} = ID$. We see that $cut^2(G')$ reaches a fixed point after two iterations:

cut^2	ID	G_1'
(0)	AB	same as left
(1)	$\{S \Rightarrow T_{cons}(AB, ID)\}$	same as left
(2)	$\{S \Rightarrow T_{cons}(AB, ID)\}$	same as left

In computing $cut^2(\{S \Rightarrow T_{cons}(AB, AB)\})$, we have $G_1 = \{S \Rightarrow T_{cons}(AB, AB)\}$; we obtain $G_1' = \{S \Rightarrow T_{cons}(AB, AB),\ S \Rightarrow S'\} = \{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$. We see that $cut^2(G_1')$ reaches a fixed point $\{S \Rightarrow T_{cons}(AB, AB) \mid T_{cons}(AB, S)\}$ after three iterations.

Approximating resulting projections after each iteration. To guarantee that all needed $f^i(G)$'s stabilize after a finite number of iterations, we use a widening operation. The idea of widening was first proposed by Cousot and Cousot [12]. 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

$$\begin{array}{l} f_0^{i_0\,\triangledown(0)}(G_0) = AB, \text{ and} \\ f_0^{i_0\,\triangledown(k+1)}(G_0) = f_0^{i_0\,\triangledown(k)}(G_0) \triangledown f_0^{i_0\,(k+1)}_{\triangledown}(G_0), \end{array} \tag{20}$$

where $f_0^{i_0\,(k+1)}_{\triangledown}(G_0)$ is computed in an on-demand fashion in the same way as $f_0^{i_0(k+1)}(G_0)$ except using the values of $f^{i\,\triangledown(k)}$'s rather than $f^{i(k)}$'s, until all needed $f^{i\,\triangledown}(G)$'s 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, 48] or explicitly [14] for regular tree grammars. The idea is to enforce the use of deterministic regular tree grammars, *i.e.*, grammars that do not produce $N \overset{s}{\Rightarrow} c^n(N_1, ..., N_n)$ and $N \overset{s}{\Rightarrow} c^n(N_1', ..., N_n')$ with $N_i' \neq N_i$ for some i. For grammars with productions only of the forms in (4), 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, while trying to use the intended widening operation proposed by Cousot and Cousot [14], we found that it does not satisfy the second property above. We describe below an appropriate widening operation that we have developed.

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 with productions only of the forms in (4).

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 productions $N \Rightarrow T_c(N_{i1}, ..., N_{in})$ for $i = 1..m$ where $m > 1$ with production $N \Rightarrow T_c(N_1, ..., N_n)$ and 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}, then replace all its occurrences in all productions by N_j. If $N_j \Rightarrow AB_{tag}$, replace all occurrences of 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 untagged AB may be simplified away, but an AB_{tag} is 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 occur 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.

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

Example 5.2 For the *odd* and *even* example, we compute

$$\begin{array}{l} odd^{1\,\triangledown(k+1)}(ID) = odd^{1\,\triangledown(k)}(ID) \triangledown odd^{1\,(k+1)}_{\triangledown}(ID), \\ even^{1\,\triangledown(k+1)}(ID) = even^{1\,\triangledown(k)}(ID) \triangledown even^{1\,(k+1)}_{\triangledown}(ID). \end{array}$$

They stabilize after four iterations. The widenings for computing $odd^{1\,\triangledown(k+1)}(ID)$ are given in detail.

	$odd^1{}_{\triangledown}(ID)$	$even^1{}_{\triangledown}(ID)$	$odd^{1\,\triangledown}(ID)$	$even^{1\,\triangledown}(ID)$
(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: $\{S \Rightarrow AB_o \mid T_{nil} \mid T_{cons}(ID, N_1),$ $N_1 \Rightarrow T_{nil} \mid T_{cons}(AB, AB_o)\}$	$\{S \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(AB, M_1),$ $M_1 \Rightarrow T_{nil} \mid T_{cons}(ID, AB_e)\}$
(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: $\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$	$\{S \Rightarrow AB_e \mid T_{nil} \mid T_{cons}(ID, M_4),$ $M_4 \Rightarrow T_{nil} \mid T_{cons}(ID, S)\}$
(4)	...	...	$\{S \Rightarrow T_{nil} \mid T_{cons}(ID, N_4),$ $N_4 \Rightarrow T_{nil} \mid T_{cons}(AB, S)\}$	same as above

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 in Step 2 force the values of recursive calls to be approximated using existing nonterminals.

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 using only themselves. If we associate N_1 with $odd^1(ID)$ and N_2 with $even^1(ID)$, then we obtain:

$$odd^1(ID) = \{N_1 \Rightarrow T_{nil} \,|\, T_{cons}(ID, N_2),\ N_2 \Rightarrow T_{nil} \,|\, T_{cons}(AB, N_1)\}$$
$$even^1(ID) = \{N_2 \Rightarrow T_{nil} \,|\, T_{cons}(AB, N_1),\ N_1 \Rightarrow T_{nil} \,|\, T_{cons}(ID, N_2)\}$$

To prove this, assume 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. This result allows us to obtain the least fixed point without the iterative computation. Our widening operation gives precisely this least fixed point too.

Approximating resulting projections within each iteration. To make the analysis 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 productions $N \Rightarrow Z$ for all Z with productions $N \Rightarrow R$ for all R such that $N \stackrel{s}{\Rightarrow} R$. Replace productions $N \Rightarrow T_c(N_{i1}, ..., N_{in})$ for $i = 1, 2$ with production $N \Rightarrow T_c(N_1, ..., N_n)$ and 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. Note that this is also the order for the size of the resulting grammar. So, this algorithm is optimal in this sense.

Example 5.3 Suppose grammar G_1 projects every second ($2n$th) element in a list, and grammar G_2 projects every third ($3n$th) element:

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

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 as below (where index ij indicate iteration i and step j, we obtain $G = G_1 \forall G_2$, which never uses the $6n+1$th and the $6n+5$th elements.

ij	combine	obtain
11	$S \Rightarrow S_1 \Rightarrow T_{nil}$ $S \Rightarrow S_1 \Rightarrow T_{cons}(AB, A)$ $S \Rightarrow S_2 \Rightarrow T_{nil}$, $S \Rightarrow S_2 \Rightarrow T_{cons}(AB, A_1)$	$S \Rightarrow T_{nil}$, $B \Rightarrow A$, $S \Rightarrow T_{cons}(AB, B)$, $B \Rightarrow A_1$
21	$B \Rightarrow A \Rightarrow T_{nil}$ $B \Rightarrow A \Rightarrow T_{cons}(ID, S_1)$ $B \Rightarrow A_1 \Rightarrow T_{nil}$, $B \Rightarrow A_1 \Rightarrow T_{cons}(AB, A_2)$	$B \Rightarrow T_{nil}$, $C \Rightarrow S_1$, $B \Rightarrow T_{cons}(ID, C)$, $C \Rightarrow A_2$
31	$C \Rightarrow S_1 \Rightarrow T_{nil}$ $C \Rightarrow S_1 \Rightarrow T_{cons}(AB, A)$ $C \Rightarrow A_2 \Rightarrow T_{nil}$, $C \Rightarrow A_2 \Rightarrow T_{cons}(ID, S_2)$	$C \Rightarrow T_{nil}$, $D \Rightarrow A$, $C \Rightarrow T_{cons}(ID, D)$, $D \Rightarrow S_2$
41	$D \Rightarrow A \Rightarrow T_{nil}$ $D \Rightarrow A \Rightarrow T_{cons}(ID, S_1)$ $D \Rightarrow S_2 \Rightarrow T_{nil}$, $D \Rightarrow S_2 \Rightarrow T_{cons}(AB, A_1)$	$D \Rightarrow T_{nil}$, $E \Rightarrow S_1$, $D \Rightarrow T_{cons}(ID, E)$, $E \Rightarrow A_1$
51	$E \Rightarrow S_1 \Rightarrow T_{nil}$ $E \Rightarrow S_1 \Rightarrow T_{cons}(AB, A)$ $E \Rightarrow A_1 \Rightarrow T_{nil}$, $E \Rightarrow A_1 \Rightarrow T_{cons}(AB, A_2)$	$E \Rightarrow T_{nil}$, $F \Rightarrow A$, $E \Rightarrow T_{cons}(AB, F)$, $E \Rightarrow A_2$
61	$F \Rightarrow A \Rightarrow T_{nil}$ $F \Rightarrow A \Rightarrow T_{cons}(ID, S_1)$ $F \Rightarrow A_2 \Rightarrow T_{nil}$, $F \Rightarrow A_2 \Rightarrow T_{cons}(ID, S_2)$	$F \Rightarrow T_{nil}$, $G \Rightarrow S_1$, $F \Rightarrow T_{cons}(ID, G)$, $G \Rightarrow S_2$,
62	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:

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

We can now use $\forall$ in place of $\vee$ everywhere in our analysis. Since $G_1 \vee G_2 \leq G_1 \forall G_2$, the sufficiency conditions still hold. Moreover, even though $G_1 \forall G_2$ may be a less precise grammar than $G_1 \vee G_2$ when used within one iteration, 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 $G_1 \forall G_2$ operation. To see this, suppose we compute a transformer application and start with AB. If we obtain $G = G_1 \forall G_2$ from the first iteration, where G_1 and G_2 are as in Example 5.3 above, then widening $AB \nabla G$ returns G. If, on the other hand, we obtain $G' = G_1 \vee G_2$ as defined in Section 3, then widening $AB \nabla G'$ returns $\{S \Rightarrow T_{nil} \,|\, T_{cons}(AB, N), N \Rightarrow T_{nil} \,|\, T_{cons}(ID, N)\}$.

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 very precise analysis results. The resulting grammars are not from any fixed abstract domain; they are based on the argument projections and the program structures.

6 Discussions

Our analysis framework for recursive data is general and powerful. The analysis has many applications and can be extended to handle other language features. It can be implemented using efficient algorithms.

Applications and extensions. *Typing.* Our analysis infers a kind of type information that helps understand and check programs. 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 information also indicates dead data whose type is 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 [54], and it helps debug and reuse program pieces. It can also be regarded as a kind of program specialization with respect to program output [46].

Dead-code elimination [3] is a straightforward application of our work. Based on how much of the return value of a function f is live, we can analyze not only how much of each parameter of f is live, as described in Section 4, but also how much of each subexpression in the definition of f is live, in a similar way. Then dead-code elimination can simply replace all subexpressions that are completely dead with $\_$.

Deforestation and *fusion* [52, 8] combine function applications to avoid building large intermediate results. To guarantee that the optimization can be done effectively, the functions and subexpressions must satisfy certain conditions, *e.g.*, be in blazed treeless form [52]. Our analysis helps identify and eliminate functions and subexpressions that do not satisfy these conditions, thus making deforestation more widely applicable.

Incrementalization, *finite differencing*, and *strength reduction* [9, 39, 34, 32] focus on replacing subcomputations whose values can be retrieved from the result of a previous computation; they can achieve asymptotic speedup. Dead-code elimination is used as the last step to remove computations whose values were used only in the replaced subcomputations. This is crucial for the speedup.

Static caching [33] caches all intermediate results in a loop body, uses them from one iteration to the next, and prunes out those that are not useful. Our analysis is needed for pruning and is crucial for reducing the space consumption. We have used this cache-and-prune method in deriving a collection of dynamic programming programs found in standard texts [2, 43, 11].

Memory allocation and compile-time garbage collection. In high-level languages with automatic memory management, an important compiler optimization is to reduce run-time overhead by reducing the memory allocated and collected. Such optimization heavily depends on analyzing various dependencies on program data, especially recursive data [26, 27, 41, 44, 17, 6]. Our analysis and framework can be used for these analyses.

Efficient implementation of lazy functional languages. Strictness analysis identifies arguments of functions that are necessarily evaluated so that we can evaluate them immediately rather than building data structures to be evaluated later [24, 51, 53]. While dead-code analysis looks for the minimum sufficient information for producing an output, strictness analysis looks for the maximum necessary information. Our analysis framework can be applied to strictness analysis by using the dual, in particular, using $\wedge$ instead of $\vee$ for combining resulting projections from different branches.

Partial evaluation. Binding-time analysis identifies computations that depend only on the static part of the input. It is a forward analysis that is equivalent to strictness analysis [31]. Analyzing partially static recursive data is important [30, 37, 16]. Again, our analysis framework can be applied.

Extensions. Our method is described here 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 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 may also obtain a finite grammar abstract domain directly from the data type information and use it to guarantee the termination of the analysis. We can also extend the analysis to handle higher-order functions, similar to extensions of other analyses to higher-order functions [13]. Finally, the analysis can be extended to handle side effects as well.

Efficient algorithms for implementations. We have shown that the three approximation operations can be performed efficiently to give very precise analysis results.

The major problem remaining is: how expensive is the ordering test used after each iteration to determine whether further iterations are needed? We have shown in Section 3 that this is decidable by reducing it to an inclusion test for regular tree grammars, since we know that inclusion test for general regular tree expressions is decidable, in particular, EXPTIME-hard [4]. In fact, inclusion test for regular tree grammars is exponential. In particular, it is at least PSPACE-complete, since inclusion test for regular string grammars is [49, 40]. Nevertheless, for *deterministic* regular tree grammars, a quadratic-time inclusion test exists [35]. Recall from Section 5 that applying widening operation after each iteration results in a deterministic grammar. Therefore, we can indeed perform grammar ordering test using the quadratic-time algorithm. Moreover, using deterministic grammars also *within* each iteration actually produces overall more precise analysis results.

These algorithms are being implemented using the Synthesizer Generator [45]. The last question left is: how many iterations are needed in the fixed-point computation? We do not have an exact characterization yet, but our experience with small examples so far is that the number is small. We suspect that it is linear in terms of the size of the program, but this needs to be verified by carefully analyzing all the grammar operations involved.

7 Related work and conclusion

Our backward dependence analysis uses domain projections to specify sufficient information. Wadler and

Hughes use projections for strictness analysis [53]. 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 [30]. It is a forward analysis equivalent to strictness analysis and uses a fixed finite abstract domain as well [31]. Mogensen, DeNiel, and others 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 [37, 16]. Another kind of analysis for recursive data is escape analysis [41, 17], but existing methods can not express as precise information as our method.

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 [27] uses necessity patterns that correspond to projections. Necessity patterns specify only heads and tails of list values. The absence analysis by Hughes [23] 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 [33] 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 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 hence for improving storage allocation. It is later used to describe other data flow information such as types and binding times [36, 37, 5, 16, 48, 46]. In particular, the analysis for backward slicing by Reps and Turnidge [46] 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 the 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.

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 [14]. We extend the grammars and handle ID and AB specially in grammar ordering, equivalence, and approximation operations. We combine carefully designed approximation operations to produce significantly more precise analysis results than previous methods. Such operations are difficult to design. While regular-tree-grammar-based program analysis can be reformulated as set-constraint-based analysis [21, 22, 14], we do not know any work that treats precise and efficient dependence analysis for recursive data as we do.

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 [42]. 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 can be applied to methods across classes without inlining.

Acknowledgments

The author would like to thank Scott Stoller and Byron Long for many helpful discussions about this work. The author is also grateful to Alex Aiken, Chris Colby, Dexter Kozen, Ming Li, and Tom Reps for discussions about related issues.

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, Mass., 1974.

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

[4] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Proceedings of the 5th International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 427–447. Springer-Verlag, Berlin, Aug. 1991.

[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 Principles of Programming Languages.* ACM, New York, Jan. 1991.

[6] 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.

[7] 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.

[8] W.-N. Chin. Safe fusion of functional expressions. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 11–20. ACM, New York, June 1992.

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

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

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

[12] 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 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, New York, Jan. 1977.

[13] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112. IEEE Computer Society Press, May 1994.

[14] 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.

[15] 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. and Syst.*, 13(4):451–490, Oct. 1991.

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

[17] 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.

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

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

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

[24] R. J. M. Hughes. Strictness detection in non-flat demains. In N. Jones and H. Ganzinger, editors, *Proceedings of the Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Oct. 1985.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, N.J., 1993.

[26] 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.

[27] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In FPCA 1989 [18], pages 54–74.

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

[29] 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.

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

[31] 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.

[32] 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.

[33] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. and Syst.*, 20(2), March 1998.

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

[35] B. Long. An algorithm for comparing deterministic regular tree grammars. Technical Report TR 503, Computer Science Department, Indiana University, Bloomington, Indiana, Feb. 1998.

[36] 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.

[37] T. Mogensen. Separating binding times in language specifications. In FPCA 1989 [18], pages 12–25.

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

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

[40] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.

[41] Y. G. Park and B. Goldber. 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.

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

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

[44] T. Reps. Shape analysis as a generalized path problem. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–11. ACM, New York, June 1995.

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

[46] 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.

[47] 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.

[48] 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.

[49] L. J. Stockmeyer and A. R. Meyer. Word problems requiring expoenetial time. In *Conference Proceedings of the 5th Annual ACM STOC*, pages 1–9. ACM, New York, 1973.

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

[51] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. E. Horwood, Chichester; Halsted Press, New York, 1987.

[52] 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.

[53] 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.

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