A Static Slicing Method for Functional Programs and Its Incremental Version

Prasanna Kumar K.  
Indian Institute of Technology, Bombay  
Mumbai, India  
prasannak@cse.iitb.ac.in

Amey Karkare  
Indian Institute of Technology, Kanpur  
Kanpur, India  
karkare@cse.iitk.ac.in

Amitabha Sanyal  
Indian Institute of Technology, Bombay  
Mumbai, India  
as@cse.iitb.ac.in

Saswat Padhi  
University of California, Los Angeles  
Los Angeles, United States of America  
padhi@cs.ucla.edu

ABSTRACT
An effective static slicing technique for functional programs must have two features. Its handling of function calls must be context sensitive without being inefficient, and, because of the widespread use of algebraic datatypes, it must take into account structure transmitted dependences. It has been shown that any analysis that combines these two characteristics is undecidable, and existing slicing methods drop one or the other. We propose a slicing method that only weakens (and not entirely drop) the requirement of context-sensitivity and that too for some and not all programs.

We then consider applications that require the same program to be sliced with respect to several slicing criteria. We propose an incremental version of our slicing method to handle such situations efficiently. The incremental version consists of a one time pre-computation step that uses the non-incremental version to slice the program with respect to a fixed default slicing criterion and processes the results to a canonical form. Presented with a slicing criterion, a low-cost incremental step uses the results of the pre-computation to obtain the slice.

Our experiments with a prototype incremental slicer confirms the expected benefits—the cost of incremental slicing, even when amortized over only a few slicing criteria, is much lower than the cost of non-incremental slicing.

CCS CONCEPTS
- Software and its engineering → Functional languages; Abstract data types; Procedures, functions and subroutines; Software testing and debugging;

KEYWORDS
Functional Programming, Static Program Analysis, Dependence Analysis, Program Slicing, Incremental Analysis

ACM Reference Format:

1 INTRODUCTION
Slicing refers to the class of techniques that delete parts of a given program while preserving certain desired behaviors, for example parts of the output. These behaviors are called slicing criteria. Applications of slicing include debugging (root-cause analysis), program specialization, parallelization and cohesion measurement.

The core of any slicing technique is dependence analysis. In the context of functional programs, it can be posed to answer the question: Given that we are interested in a certain part of the output of a program, what parts of the values of each expression in the program does it depend on. The slice is obtained by deleting expressions that do not contribute to the designated part of the output.

To be effective, the computed slice should be as precise as possible. With function calls as the primary construct in functional programs, precision requires a dependence analysis that is context-sensitive. Further, the abundant use of algebraic datatypes (e.g., lists) requires that the analysis should precisely capture structure transmitted data dependence. For example, it should be able to conclude that the value of \((\text{car} (\text{cons} x y))\) does not depend on the value of \(y\). This is not easy, if the constructor and the selector were more widely separated, say by several function calls. Unfortunately, any analysis that is context-sensitive and captures structure transmitted dependence turns out to be undecidable [15]. To get around the problem of undecidability, slicing methods either drop the requirement of context-sensitivity [16], or precise structure transmitted dependence [21].

We first propose a static slicing method for a first-order subset of the Scheme language with tuples and lists as the only algebraic data types. The slicing criteria is a regular grammar that represent sets of prefix-closed strings of the selectors \text{car} and \text{cdr}. The slicing criterion represents the part of the output of the program in which we are interested, and we view it as a demand on the program. The core of our slicing method is a dependence analysis that propagates the demand on the output to all expressions constituting the program, and a deletion phase that elides expressions that

*This work was carried out when the author was affiliated with IIT Bombay.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

CC ’19, February 16–17, 2019, Washington, DC, USA  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6277-1/19/02 ... $15.00  
https://doi.org/10.1145/3302516.3307345
We propose a slicing method based on a dependence analysis. An expression is retained in the slice if the intersection of the slicing criterion is non-empty. Given a slicing criterion, an expression is retained in the slice if the intersection of the slicing criterion with the automaton corresponding to the expression is non-empty.

The main contributions of this paper are:

1. We propose a slicing method based on a dependence analysis (Section 3) that is context-sensitive and also captures structure transmitted dependences. The method is both precise and efficient. The precision of the analysis is due to context-sensitivity, albeit in a weakened form to avoid undecidability, and the efficiency is by avoiding repeated analysis of the function body through the use of function summaries. The difficulty of creating function summaries, especially when the domain of analysis is unbounded, has been pointed out in [16].

2. We present a formal specification of dependence analysis problem itself in terms of a non-standard operational semantics. This has helped us in proving some results which are stated in the paper without proofs, for example, an independent proof of the undecidability of dependence analysis and the soundness of our approximate analysis. In addition, we have also proven the correctness of the incremental slicing algorithm with respect to the non-incremental version.

3. Our formulation allows us to derive an incremental version of the slicing algorithm by factoring out common computations while slicing using any slicing criteria (Section 5) and by reusing these computations. To the best of our knowledge, this form of incremental slicing has not been attempted before.

4. We have implemented a prototype slicer for a first-order version of Scheme (Section 6). We have also extended the implementation to higher-order programs by converting such programs to first-order using a technique called firstification [8]. The output of the slicer on the first-order programs are mapped on to the higher-order version. Experiments confirm the expected benefits of incremental slicing—the incremental step is one to four orders of magnitude faster than the non-incremental version.

2 THE TARGET LANGUAGE

Figure 2 shows the syntax of the target language with Scheme-like syntax. For ease of presentation, we restrict the language to Administrative Normal Form (ANF) [19]. In this form, the arguments to functions can only be variables. To avoid dealing with scope-shadowing, we assume that all variables in a program are distinct. Neither of these restrictions affects the expressibility of our language. We may sometimes use a label $\pi$ to refer to an expression $e$, such as $\pi : e$. However the label is not part of the language. To keep the description simple, we shall assume that each program has its own unique set of labels. In other words, a label identifies both a program point and the program that contains it.

A program in our language is a collection of function definitions followed by a main expression denoted as $e_{\text{main}}$. Applications
(denoted by the syntactic category \textit{App}) consist of functions or operators applied to variables. Constants are regarded as 0-ary functions. Expressions (\textit{Expr}) are either an \texttt{if} expression, a \texttt{let} expression that evaluates an application and binds the result to a variable, or a \texttt{return} expression. The \texttt{return} keyword is used to mark the end of a function.

3 \ DEPENDENCE ANALYSIS

We represent the selector \texttt{car} by 0 and \texttt{cdr} by 1 for brevity. An access path is a sequence of selectors, including the empty sequence. Given the value \(v\) of an expression, say \((\text{cons}\ (\text{cons} \ 1 \ 2)\ 3)\), the substructure \(v'\) corresponding to \((\text{cons} \ 1 \ 2)\) with respect to the root of \(v\) can be represented by the set of access paths \([0, 00, 01]\)—the access path 0 to reach the cons cell forming the root of \(v'\), and 00 and 01 to reach the leaves 1 and 2. We call a set of access paths a demand and use it to denote parts of a structure. Access paths are denoted by \(\sigma\) and demands by \(\delta\). Given two access paths \(\sigma_1\) and \(\sigma_2\), we use the juxtaposition \(\sigma_1\sigma_2\) to denote their concatenation. We extend this notation to a concatenate a pair of demands and even to the concatenation of a symbol with a demand: \(\sigma_1\sigma_2\) denotes the demand \(\{\alpha_1, \alpha_2\} = (\alpha_1 \in \sigma_1\) and \(\alpha_2 \in \sigma_2\}\) and \(\sigma\sigma\) is a shorthand for \(\{\alpha\} = (\alpha \in \sigma\). We use the empty sequence of selectors \(\epsilon\) to stand for the root of a structure. Thus base values such as integers have \(\epsilon\) as the only substructure.

At the heart of slicing is dependence analysis which answers the following question of a program \(P\): Given a focus expression \(e\) in \(P\), a demand \(\sigma\) representing the parts of interest in the value of \(e\), and any other expression \(e'\), what parts \(\sigma'\) of the value of \(e'\) would be required to compute \(\sigma\) of \(e\). While our analysis can answer the question at this level of generality, for this paper the focus expression will be the main expression \(e_{\text{main}}\) with a demand denoted by \(\epsilon_{\text{main}}\).

As an example, consider the demand \([10]\) on the expression \((\text{cons}\ x\ y)\). Clearly, the computation of \([10]\) part of \((\text{cons} \ x\ y)\), requires \([0]\) of \(y\). We can thus think of \((\text{cons} \ x\ y)\) as a demand transformer transforming the demand \([10]\) on itself to demands on its subexpressions—\([0]\) on \(y\) and the empty demand \(\emptyset\) on \(x\), indicating that no part of \(x\) is required.

3.1 Demand Guided Semantics

To formally specify dependence analysis, we define a small-step operational semantics called Demand Guided Semantics (DGS). Starting with a demand \(\delta_{\text{main}}\) on the main expression of a program \(P\), DGS computes the demand \(\delta\) on each expression \(e\), as it comes up for evaluation. \(\delta\) represents extent to which \(e\) has to be evaluated to meet the required demand \(\delta_{\text{main}}\) on the main expression. We call the demand thus propagated to an expression as dynamic demand and denote it as \(\delta\) to differentiate it from static demand \(\sigma\).

A key aspect of DGS is that an expression is evaluated only if the dynamic demand on it is non-empty. This is very similar to lazy evaluation [13] except that the extent of evaluation of the main expression is determined by an externally supplied demand \(\delta_{\text{main}}\). For each expression in the program, dependence analysis computes the aggregate of all dynamic demands on the expression, over all DGS traces of \(P\).

We now specify the domains used in the semantics:

\[
\begin{align*}
& v : \text{Val} = \mathbb{N} + \{\text{nil}\} + \mathbb{N} \times \mathbb{N} & - & \text{Evaluated} \\
& d : \text{Data} = \text{Val} + \text{Clo} & - & \text{Data Value} \\
& c : \text{Clo} = (\text{App} \times \text{Env}) & - & \text{Closure} \\
& \rho : \text{Env} = \text{Var} \to \text{Data} & - & \text{Environment} \\
\end{align*}
\]

An evaluated value is either a number or an empty list or a \texttt{cons} cell. A data value \(d\) may either be an evaluated value or a closure \((s, \rho)\) in which \(s\) is an application, and \(\rho\) maps free variables of \(s\) to data values. An environment is a mapping from the set of variables \(\text{Var} to \text{Data}\). The notation \([y \mapsto \rho(x)]\) represents an environment that maps the formal arguments \(y_i\) to the bindings of the actual arguments \(x_i\), \(\rho \circ \rho'\) represents the environment \(\rho'\) shadowed by \(\rho\) and \([\rho]_{x}', (\rho)_{x}'\) represents the environment \(\rho'\) restricted to the variables in the set \(X\). Finally \(FV(s)\) represents the free variables in the application \(s\).

The DGS of our language is shown in Figure 3. The semantics is given by transitions of the form \(P, s, e, \delta \rightarrow P', s', e', \delta'\). Here \(S\) is a stack of continuation frames, \(e\) is the current expression being evaluated, and \(\delta\) is the dynamic demand on \(e\). Each continuation frame is a 4-tuple \((\rho, s, e_{\text{next}}, \delta)\), signifying that the variable that is bound to the expression being evaluated is \(x\), and \(\delta_{\text{next}}\) is the next expression to be evaluated in an environment \(\rho\) and with demand \(\delta\). The initial state of the transition system is \(([]\rho, s, [\epsilon], \delta_{\text{main}}, \epsilon_{\text{main}}, (\epsilon))\). Notice the empty initial environment \([\epsilon]\rho\), and the initial stack that has a single continuation frame. In this frame, \(\epsilon_{\text{main}}\) is a distinguished variable that will eventually be bound to the value of \(\epsilon_{\text{main}}\) and \((\text{print})\) will be picked next for execution. The evaluation of \(\epsilon_{\text{main}}\) to the extent \(\delta_{\text{main}}\) is mediated through the \texttt{print} function that is external to DGS, and can trigger DGS evaluations a number of times. Each time, DGS evaluates the current expression to WHNF (represented by the demand \([\epsilon]\rho\)) and passes the result to \texttt{print}. Evaluation stops if the extent of evaluation of the expression already satisfies its dynamic demand. Else \texttt{print} initiates separate DGS evaluations for the subexpressions of the current expression with appropriately modified demands. This is a variation of a standard procedure in lazy evaluation [13].

We now briefly explain the DGS rules shown in Figure 3. Both \texttt{let} and function calls follow rules of lazy evaluation. Closures are created by \texttt{let} expressions (\texttt{let}) and evaluated at two places in a function body—while checking an \texttt{if} condition (\texttt{if} \texttt{-clo}) and at a \texttt{return} (\texttt{return} \texttt{-clo}). As an example of closure evaluation, we explain the rules for \texttt{car} and \texttt{cons}. If the demand \(\delta\) on \((\text{car} \ x)\) is \(\emptyset\) then it is not evaluated at all (\texttt{no-eval}). However, if \(\delta\) is non-empty, the context of \((\text{car} \ x)\) first does a \texttt{car}-selection on the value of \(x\) and then traverse the paths given by \(\delta\). This gives the demand on \(x\) as \([0\delta]\).

In a well typed program, the evaluation of \(x\) should result in a closure say \((\text{cons} \ y \ z)\). The DGS semantics now uses the \texttt{car-cons} rule to select \(y\) for evaluation with the propagated demand obtained by stripping off the leading \(0\) from all strings in \(\emptyset\). This gives back \(\delta\) as the demand to be propagated to \(y\). However, if the surrounding context of \((\text{cons} \ y \ z)\) had been a \texttt{cdr} instead of \texttt{car}, then this would have resulted in \(0\) on \(y\). This precision is the outcome of handling structure transmitted dependences. We now formally define the dependence analysis problem as follows:
Table: Demand guided execution semantics.

<table>
<thead>
<tr>
<th>Premise</th>
<th>Transition</th>
<th>Rule name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta$ is $\emptyset$</td>
<td>$\rho(p', y, e', \delta') : S, e, \delta \Rightarrow p', S, e', \delta'$</td>
<td>NO-EVAL</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, x, \delta \Rightarrow p', S, s, \delta$</td>
<td>VAR</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, x, \delta \Rightarrow p', S, x, \delta$</td>
<td>CAR</td>
</tr>
<tr>
<td>$\rho, S, \text{(cdr } x\text{)}, \delta \Rightarrow p, S, x, 0_{\delta}$</td>
<td>CDR</td>
<td></td>
</tr>
<tr>
<td>$\rho, S, \text{(cons} x y\text{), } (e) \Rightarrow p'$</td>
<td>PRIM-CONS</td>
<td></td>
</tr>
<tr>
<td>$\delta' = [\alpha</td>
<td>0_{\alpha} \in \delta]$</td>
<td>$\rho, S, \text{(cons} x y\text{), } \delta \Rightarrow p, S, x, \delta'$</td>
</tr>
<tr>
<td>$\delta' = [\alpha</td>
<td>1_{\alpha} \in \delta]$</td>
<td>$\rho, S, \text{(cons} x y\text{), } \delta \Rightarrow p, S, y, \delta'$</td>
</tr>
<tr>
<td>$\rho(x), \rho(y) \in \mathbb{N}$</td>
<td>$\rho, (\rho', \rho, \delta') : S, (+ x y), \delta \Rightarrow \rho', (\rho, (\rho(x), \rho(y))), S, e, \delta'$</td>
<td>PRIM-ADD</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, (+ x y), \delta \Rightarrow p, (p, x, (+ x y), \delta) : S, x, \delta$</td>
<td>PRIM-1-CLO</td>
</tr>
<tr>
<td>$\rho(y)$ is $(s, p')$</td>
<td>$\rho, S, (+ x y), \delta \Rightarrow p, (p, y, (+ x y), \delta) : S, y, \delta$</td>
<td>PRIM-2-CLO</td>
</tr>
<tr>
<td>$f$ defined as (define (f t) e)</td>
<td>$\rho, S, (f t), \delta \Rightarrow (\rho(x), S, e, \delta)$</td>
<td>FUNCALL</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, (let \ x \leftarrow \ s \ in \ e), \delta \Rightarrow \rho \cup {x \mapsto (\rho(x), S, e, \delta)$</td>
<td>LET</td>
</tr>
<tr>
<td>$\rho(x) \in \mathbb{N} &amp; \rho(x) \neq 0$</td>
<td>$\rho, S, (if \ x \ e_1 \ e_2), \delta \Rightarrow \rho, S, e_1, \delta$</td>
<td>IF-TRUE</td>
</tr>
<tr>
<td>$\rho(x) \in \mathbb{N} &amp; \rho(x) = 0$</td>
<td>$\rho, S, (if \ x \ e_1 \ e_2), \delta \Rightarrow \rho, S, e_2, \delta$</td>
<td>IF-FALSE</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, (if \ x \ e_1 \ e_2), \delta \Rightarrow \rho, (p, x, (if \ x \ e_1 \ e_2), \delta) : S, x, \delta$</td>
<td>IF-CLO</td>
</tr>
<tr>
<td>$\rho(x)$ is $(s, p')$</td>
<td>$\rho, S, (return \ x), \delta \Rightarrow \rho, S, x, \delta$</td>
<td>RETURN-CLO</td>
</tr>
</tbody>
</table>

Figure 3: Demand guided execution semantics. NO-EVAL has precedence over all rules.

Definition 3.1. The dependence analysis problem is to find an algorithm which, given a program with emain as the main expression, a demand $\delta_{main}$, a control point $\pi$ and a string $w \in \{0, 1\}^*$, outputs yes if there exists a DGS trace of emain with $\delta_{main}$ in which the expression at $\pi$ appears with a dynamic demand $\delta$ containing $w$, and no otherwise.

Theorem 3.2. The dependence analysis problem is undecidable.

While this has been proved in [15] in a slightly different setting, we have an independent proof based on our formulation of the problem that uses a completely different reduction strategy. Note that the Demand-Guided Semantics is only a formal mechanism to specify (not compute) dependences. The actual computation of dependences is based on an analysis that we describe in 3.2.

3.2 Dependence Analysis

Given Theorem 3.2, our analysis will eventually compute an over-approximation of the actual dependences (Section 4.1). For now we attempt an exact analysis shown in Figure 4. Given an application $s$ and a demand $\delta$, $A$ returns a demand environment that maps arguments of $s$ to their demands. The third parameter to $A$, denoted $DS$, represents context-independent summaries of the functions in the program and is used to analyze function calls. This will be explained shortly.

For all applications except function calls, the static propagation of demands bears close resemblance to the propagation of demands in their corresponding DGS rules. The ideas behind the car and cons rules have already been discussed in the earlier section. Since (null? $x$) only requires the root of $x$ to examine the constructor, a non-null demand on (null? $x$) translates to the demand $e$ on $x$. A similar reasoning also explains the rule for (+ $x$ $y$). Since, both $x$ and $y$ evaluate to integers in a well typed program, a non-null demand on (+ $x$ $y$) translates to the demand $e$ on both $x$ and $y$.

Just as $A$ defines how a primitive like car maps a demand on itself to demands on its arguments, we would like to derive a similar transformation for user-defined functions. Since user-defined functions are, in general, mutually dependent, we define this transformation simultaneously for all user-defined functions. This is given by the inference rule $\delta\Rightarrow\text{DS}^f_{\delta}$, defining how a demand $\sigma$ on a call to $f$ is propagated to its rth parameter. The rule for function calls uses $DS$ to propagate demands to the arguments of a specific call. We look upon the functions for $DS_f$ as a context-independent summary of $\delta\Rightarrow\text{DS}^f_{\delta}$—independent because it is parameterized with respect to the demand that will be instantiated at the place where the function is called.

The rule $\delta\Rightarrow\text{DS}^f_{\delta}$ specifies the fixed-point property to be satisfied by DS, namely, the demand transformation assumed for each function in the program should be the same as the demand transformation calculated from the body of the function. The reader will notice the similarity between this rule and the rule for recursive lets in the Hindley-Milner system of type inference [3, 7, 14]. An operational interpretation of the rule to find $\text{DS}^f_{\delta}(\sigma)$ proceeds by analyzing $e_f$, the body of $f$, with respect to a symbolic demand $\sigma$. Then $\text{DS}^f_{\delta}(\sigma)$ is the union of the demands on all occurrences of the rth argument in $e_f$. A call to a function, say $g$, in $e_f$ is analysed using the summary $\text{DS}_g$. In general, this results in a recursive description of $\text{DS}^f_{\delta}(\sigma)$. We explain in Section 4 how to convert this to a closed form.

We finally discuss the rules for expressions given by $D$. The rule for return is obvious. The rule for if propagates any non-null demand unchanged to both the then-part and the else-part. The
variable used for the condition gets an \( \epsilon \) demand if the incoming demand is non-null as it will always evaluate to a boolean value. The transformation of the demand on the if to the demand on the conditional variable captures the traditional notion of control dependence, i.e. whether the value of the if-expression is the value of \( e_1 \) or \( e_2 \) depends on whether \( x \) is true or false. The rule for (let \( x \leftarrow s \) in \( e \)) first uses \( s \) to calculate the demand environment \( DE \) of the let-body \( e \). The demand on \( s \) is the union of the demands on all occurrences of \( x \) in \( e \). Notice that the demand environment for each expression \( e \) includes the demand on \( e \) itself apart from its subexpressions.

While the computation of function summary assumed a symbolic demand for each function, to compute the demand environment, we have to supply the concrete demand for each function. To start with, the demand environment for \( e_{\text{main}} \) is computed with the externally supplied demand \( \sigma_{\text{main}} \). Further, the demand on a function \( f \), denoted \( \sigma_f \), is the union of the demands at all call-sites of \( f \). The demand environment of a function body \( e_f \) is calculated using \( \sigma_f \). If there is a call to \( g \) inside \( e_f \), the demand summary \( DS_g \) is used to propagate the demand across the call. This completes the formal description of dependence analysis.

For the rest of the paper, we consider the program in Figure 5 as our running example. The program takes a list of integers as input and computes the minimum and maximum values along with their positions in the input list. The function \( \text{mmp} \) keeps track of the current minimum and maximum value using the arguments \( xv \) and \( nv \). It compares every element with \( xv \) and if the current element is greater, it updates \( xv \) to be the current element and processes the rest of the list. \( p \) which keeps track of the position of the current element is used to update \( xp \). The minimum value and its position are also computed similarly.

Consider \( DS_{\text{mmp}}^2 \), the function that propagates the demand on a call to \( \text{mmp} \) to its second argument. This is specified by demand-summary, and the operational interpretation of this rule requires us to do a demand analysis of the body of \( \text{mmp} \) with a symbolic demand \( \sigma \) and union the resulting demands on all occurrences of the second argument \( p \) in the body. Firstly notice that, according to the rules of if and let, the demand \( \sigma \) is propagated without change to the three calls at lines 6, 8 and 9. Further, \( p \) appears as the fourth argument to the call to \( \text{mmp} \) at line 6 and the sixth in the call to

\[
A :: \text{FuncName} \rightarrow \text{FuncSummaries} \\
\forall f, \forall s, \forall e_f \quad D(e_f, \sigma, DS) = \text{DE} = DS_f = \bigcup_{\alpha \in E} \text{DE}(\alpha) \\
\text{(Demand-Summary)}
\]

Figure 4: Dependence Analysis

Figure 5: A program to compute the min and max elements in a list along with their positions. While the program is not in ANF, this is not important for the ensuing discussion.
mmp at line 8. Clearly the demands on these two occurrences of p are DS₄ₘₜₚ(σ) and DS₆ₘₜₚ(σ). Also notice that the demands of the three occurrences of p1 at lines 6, 8 and 9 are the same, namely DS₂ₘₜₚ(σ). And since p is being used to define p1 at line 4, by the let rule, the demand on this occurrence of p is:

if (DS₂ₘₜₚ(σ) ≠ 0) then {ε} else θ

Bringing everything together, we get:

DS₂ₘₜₚ(σ) = DS₄ₘₜₚ(σ) ∪ DS₆ₘₜₚ(σ) ∪ DS₈ₘₜₚ(σ)

if (DS₂ₘₜₚ(σ) ≠ 0) then {ε} else θ

(1)

We bring this equation to a closed-form by substituting the values of DS₄ₘₜₚ(σ) and DS₆ₘₜₚ(σ) and eliminating the recursion in DS₂ₘₜₚ(σ). This is the subject of the next section. Also recollect the concrete demand σₘₜₚ used to find the demand environment of the body of mmp is the union of the demands on all calls to mmp. Now there here are four calls to mmp in the program. If we assume that the concrete demand on the body of mmp to be σₘₜₚ, then it is easy to see that this demand propagates without change to the three calls in the body of mmp. The call to mmp in main also has the demand σₘₜₚ. Thus we get:

σₘₜₚ = σₘₜₚ ∪ σₘₜₚ ∪ σₘₜₚ ∪ σₘₜₚ

(2)

This gives the value of σₘₜₚ as σₘₜₚ. Finally, the function summaries and the demands on function bodies are used to compute the demand environment for function bodies. When the body of mmp is analyzed with the demand σₘₜₚ, the demand on p1 is DS₂ₘₜₚ(σₘₜₚ). Thus, by the let rule, the demand on p at π, denoted Dₚ, is given as:

Dₚ = if (DS₂ₘₜₚ(σₘₜₚ) ≠ 0) then {ε} else θ

(3)

In summary, at the end of the analysis we shall have (i) a set of equations (e.g. 1) defining the function summaries DSₙₚ for each argument of each function, (ii) an equation (e.g. 2) for the concrete demand σₚ on the body of each function f, and (iii) equations (e.g. 3) specifying the demand Dₚ at each program point π. Our dependence analysis is sound in the following sense:

Theorem 3.3. Assume that the dependence analysis of a program P with a demand σₘₜₚ results in a demand environment in which the demand on an expression e is σ. Also consider a Dₚ trace of P with the same demand σₘₜₚ on the main expression. For any evaluation of e in the trace, if the demand on e is δ, then δ ⊆ σ.

4 Computing Dependences

We now describe how to (i) set up the equations for function summaries, DSₙₚ, and convert them into a closed form, and (ii) use these summaries to obtain the demand environment for all function bodies, indeed the entire program. Also recall that the summaries are functions that describe how a symbolic demand on a function call is propagated to the arguments of the function.

However, notice that the rules of dependence analysis require us to do operations that cannot be done symbolically. The cons rule, for example, is defined in terms of the set {α | α ∈ σ}. Clearly this requires us to know whether the strings in σ start with a θ. Similarly, the if rule requires to know whether σ is θ. The way out is to treat these operations also symbolically. For this we introduce three new symbols 0, 1 and θₖ, to capture the intended operations. If 0 represents selection using car, 1 is intended to represent its opposite—use by the constructor cons as its left argument. Thus 00 (cons-car cancellation) reduces to the empty string ε. Similarly θₖ represents the symbolic transformation of any non-null demand to ε and null demand to itself. These transformation are defined and the order in which they are applied is made deterministic through the simplification function S.

\[
S(\{\}) = |ε| \\
S(\{0\}) = εS(σ) \\
S(1σ) = |α| |α ∈ S(σ)| \\
S(θₖσ) = \begin{cases} \\
0 & \text{if } S(σ) = 0 \\
|ε| & \text{otherwise} \\
\end{cases}
\]

The symbol 0 strips the leading 0 from the string following it, as required by the rule for cons. Similarly, θₖ examines the string following it and replaces it by θ or |ε|. The rules for cons and if in terms of the new symbols are:

\[
A(π: (\text{cons} σ₁ x π₂ y), σ, DS) = \{π₁ → 0σ, π₂ → 1σ\}
\]

\[
D(π: (\text{if } π₁: x π₂ y), σ, DS) = D(ε₁, σ, DS) ∪ D(ε₂, σ, DS) ∪ \{π₁ → 0σ, π → σ\}
\]

The rules for + and null2 are modified similarly. Now the demand summaries can be obtained symbolically with the new symbols as markers indicating the operations that should be performed on the string following them. After we compute demand environments in closed-form and the slicing criterion σₘₜₚ is substituted as a concrete demand for the main expression σₘₜₚ, i.e., σₘₜₚ = σₘₜₚ, we eliminate the symbols 0, 1 and θₖ using S.

4.1 Converting the Results of Analysis to Grammars

As mentioned earlier, and illustrated in the example in the last section, to obtain the context-independent summary DSₙₚ, we start with a symbolic demand σ and compute the demand environment for eₙ, the body of f. From this we calculate the demand on the nth argument of f, say x. This is the union of demands of all occurrences of x in the body of f. The demand on the nth argument is equated to DSₙₚ(σ). Since the body may contain other calls, the dependence analysis within eₙ makes use of DS in turn. The equations shown below define DS₂ₘₜₚ(σ). Notice that the earlier equation for DS₂ₘₜₚ(σ) in Section 3.2 has been rewritten in terms of the symbols 0, 1 and θₖ.

\[
DS₂ₘₜₚ(σ) = DS₆ₘₜₚ(σ) ∪ DS₄ₘₜₚ(σ) ∪ \thetaₖDS₂ₘₜₚ(σ)
\]

\[
DS₄ₘₜₚ(σ) = 10σ ∪ DS₄ₘₜₚ(σ)
\]

\[
DS₆ₘₜₚ(σ) = 11σ ∪ DS₆ₘₜₚ(σ)
\]

\[1\text{the choice of this symbol is to remind the reader that its action depends on whether the demand following it is } 0 \text{ or not.} \]
As noted in [16], the main difficulty in obtaining a convenient function summary is to find a closed-form for $DS^2_{\text{mmp}}(\sigma)$ instead of the recursive description. Our solution to the problem lies in the following observation: Since we know that the rules of dependence analysis always prefix $\sigma$ with symbols, we can write $DS^2_{\text{j}}(\sigma)$ as $DS^2_{\text{j}}(\sigma) \cup \{ \epsilon \}$, where $DS^2_{\text{j}}$ is a set of strings over the alphabet $\{0, 1, 0, 0, 0\}$, and $\sigma$ represents the effect of $DS^2_{\text{j}}$ on $\sigma$. The modified equations after doing this substitution will be,

$$DS^2_{\text{mmp}} = DS^2_{\text{mmp}} \cup DS^4_{\text{mmp}} \cup \emptyset e DS^2_{\text{mmp}}$$

$$DS^4_{\text{mmp}} = DS^4_{\text{mmp}} \cup DS^6_{\text{mmp}} \cup \emptyset e DS^2_{\text{mmp}}$$

$$DS^6_{\text{mmp}} = DS^6_{\text{mmp}} \cup \emptyset$$. 

Notice that we can factor out and cancel $\sigma$ on both sides of the equations. Thus, what we have so far is:

$$DS^2_{\text{mmp}}(\sigma) = DS^2_{\text{mmp}}(\sigma)$$

$$DS^2_{\text{mmp}} = DS^2_{\text{mmp}} \cup DS^4_{\text{mmp}} \cup \emptyset e DS^2_{\text{mmp}}$$

$$DS^4_{\text{mmp}} = DS^4_{\text{mmp}} \cup DS^6_{\text{mmp}} \cup \emptyset e DS^2_{\text{mmp}}$$

$$DS^6_{\text{mmp}} = DS^6_{\text{mmp}} \cup \emptyset$$

Notice that the set of five unrestricted productions shown are independent of $\pi$, and the symbol $\emptyset$ represents a set of strings, and is a part of the solution. The solution of $DS^2_{\text{j}}$ maps a demand $\sigma$ to $DS^2_{\text{j}}(\sigma)$.

Figure 6: The simplification of the automaton $M_{\text{r}}^\sigma$: (a), (b) and (c) show the simplification for the slicing criterion $\sigma = \{ e, 0, 0.01, 00 \}$, while (d), (e) and (f) show the simplification for the criterion $\sigma = \{ e, 1, 0, 10, 00 \}$.

We get around the problem of undecidability by using the technique of Mohri-Nederhof [9] which takes a CFG $G$ as input and returns a strongly regular grammar $R$ that is an over-approximation of $G$. The grammar $P^\sigma_{\text{unrestricted}}$ consists of a CFG, say $M$, along with the non-CFG productions in $\text{unrestricted}$. Mohri-Nederhoff is applied only to $G$, giving a regular grammar, say $R$. The membership problem of $R \cup \text{unrestricted}$ is (efficiently) decidable. The NFA corresponding to this strongly regular grammar is denoted as $M^\sigma_{\text{r}}$. The simplification rules can be applied on $M^\sigma_{\text{r}}$ without any loss of precision. The details of the simplification process are in [5]. Currently, we do not know if there is a better approximation for $G \cup \text{unrestricted}$ with the membership question still decidable. Even if such an approximation exists, the algorithm to answer the membership question may not be efficient for practical purposes.

For our running example, the grammar after dependence analysis is already regular, and thus remains unchanged by Mohri-Nederhoff transformation. The automata in Figure 6a–c and 6d–f correspond to the two slicing criteria $\sigma_{\text{mmp}} = \{ e, 0, 0.01, 00 \}$ and $\sigma_{\text{mmp}} = \{ e, 0, 0.01, 1, 10 \}$ and illustrate the simplification of corresponding Mohri-Nederhof automata $M^\sigma_{\text{mmp}}$. It can be seen that, when the slicing criterion is $\{ e, 0, 0.01, 1, 10 \}$, the language of $D_{\pi}$ is empty and hence the argument $p$ can be sliced away. A drawback of the method outlined above is that with a change in the slicing criterion, the entire process of grammar generation, Mohri-Nederhoff approximation and simplification has to be repeated. This is likely to be inefficient for large programs.
Figure 7: (a) The canonical automaton $A_{\sigma}$ and (b) the corresponding completing automaton $\overline{A}_{\sigma}$.

5 INCREMENTAL SLICING

The incremental algorithm avoids repetition of computation when the same program is sliced with different criteria. This is done by pre-computing the part of the slice computation that is common to all slicing criteria.

Since any slicing criterion except $\emptyset$ must include the root of the result of the program, the pre-computation involves slicing with $\{\epsilon\}$. Indeed, this is the first step of the following three-step pre-computation process: (i) Using the non-incremental slicing method with the fixed slicing criterion $\{\epsilon\}$ to compute the demand at each expression $\pi$. $\epsilon$ and applying the Mohri-Nederhof procedure to construct the corresponding automaton $M_\sigma^{\{\epsilon\}}$, (ii) a step called canonicalization which applies the simplification rules on $M_\sigma^{\{\epsilon\}}$, but stops when the symbols 0 and 1 of every accepting string of the resulting automaton are only at the end, and, (iii) from the canonical automaton, constructing an automaton called the completing automaton, the output of the pre-computation step. We now explain the step called canonicalization.

For the running example, the automaton $M_\sigma^{\{\epsilon\}}$ is shown in Figure 7a. Each accepting string in this automaton has 0 and 1 symbols only at the end. Thus the automaton is canonical, and we shall denote it as $A_{\sigma}$. It is clear that if $A_{\sigma}$ is concatenated with a slicing criterion that starts with the symbol 01, the result, after simplification, will be non-empty, and the expression at $\pi$ has to be retained in the slice. We call such a string a completing string for $A_{\sigma}$. Observe that the completing string was easy to detect since the canonicalization step pushed all the 0 and 1 symbols towards the final state in the canonical automaton. Similarly, 11 is also a completing string for the same automaton.

Now consider the automaton in Figure 7b, called the completing automaton, that recognizes the language $(01 + 11)(0 + 1)^\ast$. This automaton recognizes all completing strings for $A_{\sigma}$ and nothing else. Thus for an arbitrary slicing criterion $\sigma$, it suffices to intersect $\sigma$ with the completing automaton to decide whether the expression at $\pi$ will be in the slice. In fact, it is enough for the completing automaton to recognize just the language $(01 + 11)$ instead of $(01 + 11)(0 + 1)^\ast$. The reason is that any slicing criterion, say $\sigma$, is prefix closed, and therefore $\sigma \cap (01 + 11)$ is $\emptyset$ if and only if $\sigma \cap (01 + 11)(0 + 1)^\ast$ is $\emptyset$. The incremental algorithm generalizes these observations.

For constructing the completing automaton for an expression $e$, we saw that it would be convenient to canonicalize the automaton $M_\sigma^{\{e\}}$ to an extent that all accepted strings have 0 and 1 symbols only at the end. We now give a set of rules $C$, that captures this generalization.

Algorithm 1: Functions to create the completing automaton and the slicing function.

```
Function createCompletingAutomaton(A)
Data: The Canonicalized Automaton
A = (Q, \{0, 1, 0, 1\}, \delta, q_0, F)
Result: $\overline{A}$, the completing automaton for $A$
F' = (q_0 + q_1 \in Q, \text{hasBarFreeTransition}(q_0, q_1, \sigma))
/* Reverse the 'bar' transitions: directions as well as labels */
foreach (transition $\delta(q, \bar{0}) \rightarrow q'$)
do
    add transition $\delta'(q', 0) \rightarrow q$
foreach (transition $\delta(q, 1) \rightarrow q'$)
do
    add transition $\delta'(q', 1) \rightarrow q$
q' ← new state /* start state of $\overline{A}$ */
foreach (state $q \in F$)do
    add transition $\delta'(q', \epsilon) \rightarrow q$
return $\{Q \cup \{q'\}, \{0, 1\}, \delta', q'_0, F'\}$

Function isInSlice(e, $\sigma$)
Data: expression $e$, slicing criteria $\sigma$
Result: Decides whether $\epsilon$ should be retained in slice
return $(L(\overline{A}) \cap \sigma = \emptyset)$
```

simplification.

$C(\{\epsilon\}) = \{\epsilon\}$
$C(0\sigma) = 0C(\sigma)$
$C(1\sigma) = 1C(\sigma)$
$C(0\sigma) = \{0\} \cup C(\sigma)$
$C(1\sigma) = \{1\} \cup C(\sigma)$
$C(\emptyset \sigma) = \emptyset$ \quad $C(\emptyset \sigma)$
$C(\{\epsilon\} \cup \{0, 1\} \cup \{1, 0\} \cup \{00, 01\}) = C(\emptyset \sigma)$
$C(\sigma_1 \cup \sigma_2) = C(\sigma_1) \cup C(\sigma_2)$

$C$ differs from $S$ in that it accumulates continuous runs of 0 and 1 at the end of a string. Notice that $C$, like $S$, simplifies its input string from the right. Here is an example of $C$ simplification:

$10, 0000, 01110 \xrightarrow{C} 10, 00000, 010 \xrightarrow{C} 10, 00, 010$

In contrast the simplification using $S$ gives:

$10, 00000, 01110 \xrightarrow{S} 10, 00000, 01110 \xrightarrow{S} 10, 00000, 01000 \xrightarrow{S} \ldots \xrightarrow{S} \emptyset$

$C$ satisfies two important properties:

1. The result of $C$ always has the form $(0 + 1 + \emptyset)^\ast (0 + 1)^\ast$. Further, if $\sigma \subseteq (0 + 1 + \emptyset)^\ast$, then $C(\sigma) = \sigma$.
2. $S$ subsumes $C$, i.e., $S(C(\sigma) \cup C(\sigma_1)) = S(\sigma_1 \sigma_2)$.

Note that while we have defined the canonicalization over a language, the actual canonicalization takes place over an automaton—specifically the automaton $M_\sigma$ obtained after Mohri-Nederhof transformation. The process of canonicalization over an automaton is a minor variation of the simplification process [5]. Specifically, (i) adjacent 00 and 11 edges are replaced by an $\epsilon$ edge and the resulting automaton is made deterministic, until there are no more such edges, and (ii) edges with labels 0 or 1 are retained only if their targets have a path reaching some final node, and the labels on this path consist only of 0 or 1 symbols. It is in the second step that the canonicalization differs from simplification over automata.
Algorithm 1 describes function createCompletingAutomaton that takes $A_{\pi}$, the canonical Mohri-Nederhof automaton for the slicing criterion $\{e\}$, as input, and constructs the completing automaton $\overline{A}_{\pi}$. Recollect that the strings recognized by $A_{\pi}$ are from $(0 + 1 + 0_\epsilon)^*(\overline{0} + 1)^*$. Call the set of states reachable from the start state using only edges with labels $\{0, 1, 0_\epsilon\}$ as the frontier set. The completing automaton is a copy of canonical automaton with edges labeled by $\overline{0}$ and $1$ symbols reversed, and the symbols themselves replaced by $0$ and $1$ respectively. All edges with labels $\{0, 1, 0_\epsilon\}$ are dropped. Further, all states in the frontier set are marked as final states, and a new start node is added with transitions to the states corresponding to the final states of canonical automaton.

The completing automaton is computed only once and can be reused whenever the program needs to be sliced. To decide whether $\pi \vdash e$ can be sliced out, the function inSlice described in Algorithm 1 just checks if the intersection of the slicing criteria with $L(\overline{A}_{\pi})$ is null. We now present a theorem that states that the incremental algorithm inSlice($e, \sigma$) is sound in that it produces the same result as the non-incremental version which checks whether the language after the simplification of the Mohri-Nederhof automaton is empty.

**Theorem 5.1.** $S(L(M''_\sigma)) \neq \emptyset \iff \text{inSlice}(e, \sigma)$

### 6 EXPERIMENTS AND RESULTS

In the absence of the details of implementations of other slicing methods, we have compared the incremental step of our method with the non-incremental version. Our experiments show that the incremental algorithm is better even when the overhead of constructing the completing automata is amortized over only a few slicing criteria.

Our benchmarks consist of first order programs derived from the nolfb suite [10]. The higher order programs have been handcrafted to bring out the issues related to higher order slicing. The program named parser includes most of the higher order parser combinators required for parsing. Table 1 shows the time required for slicing with different slicing criteria. For each benchmark, we first show, the precomputation time, i.e. the time required to construct the completing automata. We then consider three different slicing criteria, and for each slicing criterion, present the times for non-incremental slicing and the incremental step. Table 1 shows that for all benchmarks, the time for computing the completing automaton is comparable to the time for computing the slice non-incrementally. Since computing completing automata is a one time activity, incremental slicing is very efficient even when a program is sliced only twice. As seen in Table 1, the time taken for the incremental step is orders of magnitude faster than non-incremental slicing, confirming the benefits of reusing the completing automata.

We also show the number of expressions in the original program and in the slice produced to demonstrate the effectiveness of the slicing process itself. Here are some of the interesting cases. It can be seen that the slice for nqueens for any slicing criterion includes the entire program. This is because finding out whether a solution exists for nqueens requires the entire program to be executed. On the other hand, the program lambda is a $\lambda$-expression evaluator that returns a tuple consisting of an atomic value and a list. The criterion $\{e, 0\}$ requires majority of the expressions in the program to be present in the slice to compute the atomic value. On the other hand, the criterion $\{e\} \cup \{e, 1\}$ do not require any value to be computed and expressions which compute the constructor only are kept in the slice, hence our algorithm is able to discard most of the expressions. After examining the nature of the benchmark programs, the slicing criteria and the slices, we conclude that slicing is most effective when the slicing criterion selects parts of a bounded structure, such as a tuple, and the components of the tuple are produced by parts of the program that are largely disjoint.

### 7 RELATED WORK

Most of the efforts in slicing have been for imperative programs. The surveys [1, 20, 22] give good overviews of variants of the slicing problem and their solution techniques. We focus on previous work related to functional programs. Silva, Tamarit and Tomás [21] have proposed a slicing method for Erlang. While their method handles calling contexts precisely, as pointed out by the authors themselves, they give up on structure transmitted dependences. When given the Erlang program: $(\text{main}) \rightarrow x = [1, 2], y = x, y$, their method gives the imprecise slice $(\text{main}) \rightarrow x = [1, 2], y = x, y$ when sliced on the variable $y$. Notice that the slice retains the constant $2$ for not handling the interaction between cons and cdr. For the equivalent program $(\text{let } x \leftarrow \text{cons } 1 \text{ in } (\text{let } y \leftarrow (\text{car } x) \text{ in } y))$ with the slicing criterion $e$, our method would correctly compute the demand on the constant $2$ as $I(e \cup \emptyset)$. This simplifies to the demand $0$, and $2$ would thus not be in the slice.

The slicing technique that is closest to ours is due to Reps and Turnidge [16]. They use projection functions, represented as tree grammars, as slicing criteria. Given a program $P$ and a projection function $\psi$, their goal is to produce a program which behaves like $\psi \circ P$. Their analysis consists of propagating the projection function backwards to all subexpressions of the program. After propagation, any expression with the projection function $\perp$ (corresponding to our $\emptyset$ demand), is sliced out of the program. Liu and Stoller [6] use a similar method for dead code analysis and elimination.

These techniques differ from ours in two respects. Unlike us, they give up on context-sensitivity and merge calling contexts, thus affecting the precision of the slice. The second difference relates to our computation of function summaries using symbolic demands that enables the incremental version. Consider, as example, the program fragment $\pi : (\text{cons } \pi_1 : x : \pi_2 : y)$ representing the body of a function. Dependence analysis with the symbolic demand $\sigma$ gives the demand environment $[\pi \mapsto \sigma, \pi_1 \mapsto \emptyset, \pi_2 \mapsto 1\sigma]$. Notice that the demands $\pi_1$ and $\pi_2$ are in terms of the symbols $\emptyset$ and $1$, a result of our decision to work with symbolic demands. Slicing with the default criterion $e$ and canonicalizing the result gives the demand environment $[\pi \mapsto e, \pi_1 \mapsto \emptyset, \pi_2 \mapsto 1]$. There is enough information in this to deduce that $\pi_1 (\pi_2)$ will be in the slice only if the slicing criterion includes $\emptyset (1)$. Since the methods in [16] and [6] deal with concrete demands, it is difficult to see the techniques behind our incremental version being replayed on their methods.

There are other less related approaches to slicing. Rodrigues and Barbosa [17] use a graph based approach for component identification in Haskell programs. On a different note, Rodrigues and Barbosa [18] use program calculation for obtaining a slice. Given a program $P$ and a projection function $\psi$, they calculate a program equivalent to $\psi \circ P$. However the method is not automated. Finally,
dynamic slicing techniques have been explored by Perera et al. [12], Ochoa et al. [11] and Biswas [2].

8 CONCLUSIONS AND FUTURE WORK

We have presented an algorithm for slicing first order functional programs and also its incremental version. We introduce a notion called demand that represents parts of a structure that are of interest. We view the slicing criterion as a demand on the value of the main expression of a program, and design a dependence analysis that propagates this demand to all expressions in the program. Only the expressions with non-empty demands are retained in the slice.

Ideally, the slicing algorithm should be based on a precise dependence analysis, which could be attained (amongst other means) by: (i) making the analysis context-sensitive and avoiding analysis over infeasible paths, and (ii) by capturing constructor-selector interaction precisely and avoiding false data dependences. Requiring both, however, results in undecidability. Our approach to dependence analysis gives a context free grammar to which we add two non-context free productions to capture constructor-selector interactions. The language derived by the grammar is the precise result of the analysis. However, given the undecidability of dependence analysis, we derive an algorithm, by overapproximating the context-free part to a regular grammar. The analysis is precise in many cases—non-recursive and tail-recursive functions for example.

In the incremental version, a per program pre-computation step slices the program with the default criterion $\epsilon$. This step factors out the computation that has to be done while slicing with any slicing criterion. The result, reduced to a canonical form, can now be used to find the slice for a given criterion with minimal computation. Experiments with our implementation confirm the benefits of incremental slicing.

We believe that our work is much more complete than earlier work on slicing functional programs. We have a specification of dependence analysis problem through DGS which was used to give an alternate proof of undecidability. DGS was also used to prove the soundness of our (approximate) dependence analysis. To the best of our knowledge, this is the first attempt to put slicing on such a formal basis. We also have a proof of the correctness of the incremental method with respect to the non-incremental version. Due to lack of space the proofs could not be included. And finally, we have extended our approach to higher-order programs through firstification and reported results on a number of benchmarks.

There are however two areas of concern. While our incremental slicer is fast enough, the pre-computation step is slow, primarily because of the canonicalization step. The other concern is the imprecision that arises out of the Mohri-Nederhoff approximation. As an example, consider the function $\text{mapsq}$:

\begin{verbatim}
(define (mapsq l)
  (if (null? l) (return 1)
       (return (cons (sq (car l)) (mapsq (cdr l))))))
\end{verbatim}

The reader can verify that the actual function summary for $\text{mapsq}$ is: $\text{DS}_{\text{mapsq}}(\sigma) = \text{DS}_{\text{sq}}(\sigma) \sigma$, where $\text{DS}_{\text{sq}}$ is the language $\epsilon \mid 1^n 1^n \mid 1^n 00, 00^n$, for $n \geq 0$. Now, given a slicing criterion $\sigma = \{ \epsilon, 1, 11, 110 \}$ standing for the path to the third element of a list, it is easy to see that $\text{DS}_{\text{mapsq}}(\sigma)$ after simplification would give back $\sigma$ itself, and this is the most precise slice. However, due to Mohri-Nederhoff approximation $\text{DS}_{\text{mapsq}}$ would be approximated

<table>
<thead>
<tr>
<th>Program</th>
<th>Pre-computation time (ms)</th>
<th>#exprs in program</th>
<th>Slicing with ${\epsilon}$</th>
<th>Slicing with ${\epsilon, 0}$</th>
<th>Slicing with ${\epsilon, 1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>treetojoin</td>
<td>1817.2</td>
<td>609</td>
<td>1556.7</td>
<td>1.3</td>
<td>565</td>
</tr>
<tr>
<td>deriv</td>
<td>217.0</td>
<td>415</td>
<td>1096</td>
<td>0.9</td>
<td>267</td>
</tr>
<tr>
<td>minmaxpos</td>
<td>32.4</td>
<td>189</td>
<td>13.7</td>
<td>0.7</td>
<td>166</td>
</tr>
<tr>
<td>nperm</td>
<td>408.5</td>
<td>648</td>
<td>285.1</td>
<td>1.3</td>
<td>249</td>
</tr>
<tr>
<td>paraffins</td>
<td>1898.1</td>
<td>1268</td>
<td>1775.1</td>
<td>1.3</td>
<td>16</td>
</tr>
<tr>
<td>knightstour</td>
<td>834.1</td>
<td>675</td>
<td>647.9</td>
<td>1.4</td>
<td>511</td>
</tr>
<tr>
<td>huffman</td>
<td>4208.2</td>
<td>1124</td>
<td>3324.0</td>
<td>2.3</td>
<td>897</td>
</tr>
<tr>
<td>sudoku</td>
<td>94273.3</td>
<td>2105</td>
<td>80065.4</td>
<td>4.6</td>
<td>2070</td>
</tr>
<tr>
<td>leas</td>
<td>10144.0</td>
<td>703</td>
<td>6730.0</td>
<td>1.5</td>
<td>694</td>
</tr>
<tr>
<td>nqueens</td>
<td>165.0</td>
<td>366</td>
<td>96.8</td>
<td>0.9</td>
<td>366</td>
</tr>
<tr>
<td>linecharcount</td>
<td>17.6</td>
<td>108</td>
<td>8.8</td>
<td>0.5</td>
<td>92</td>
</tr>
<tr>
<td>studentint</td>
<td>90.6</td>
<td>310</td>
<td>647.0</td>
<td>0.6</td>
<td>106</td>
</tr>
<tr>
<td>lal</td>
<td>43.7</td>
<td>158</td>
<td>25.3</td>
<td>0.4</td>
<td>105</td>
</tr>
<tr>
<td>fibheap</td>
<td>133716.3</td>
<td>679</td>
<td>101365.1</td>
<td>2.2</td>
<td>53</td>
</tr>
<tr>
<td>lambda</td>
<td>1625.9</td>
<td>750</td>
<td>1179.2</td>
<td>1.6</td>
<td>27</td>
</tr>
<tr>
<td>Told</td>
<td>25.0</td>
<td>127</td>
<td>11.2</td>
<td>0.3</td>
<td>17</td>
</tr>
<tr>
<td>barneshut</td>
<td>7859.1</td>
<td>1600</td>
<td>5876.2</td>
<td>2.8</td>
<td>149</td>
</tr>
<tr>
<td>mappal</td>
<td>20.3</td>
<td>102</td>
<td>13.8</td>
<td>0.6</td>
<td>58</td>
</tr>
<tr>
<td>sssp</td>
<td>2913.9</td>
<td>772</td>
<td>2260.9</td>
<td>1.3</td>
<td>385</td>
</tr>
<tr>
<td>parser</td>
<td>3711.4</td>
<td>1294</td>
<td>2473.1</td>
<td>1.1</td>
<td>178</td>
</tr>
</tbody>
</table>
We provide the detailed proof of the correctness of incremental δ. Next we discharge the assumption regarding applications by for evaluation with a context, say δ on an expression the problem of whether e belongs to an arbitrary grammar in the class is undecidable. This is done by reducing the halting problem to the to the e-recognition problem above. We then consider a subset of CG, say CG′, that is large enough to replay the undecidability proof. Specifically, the set CG′ represents encoding of all Turing Machines. Finally, we show that any grammar G in C can be converted to a program P such that the problem of whether e belongs to L(G) can be reduced to the dependence analysis problem.

A.1 Undecidability of Dependence Analysis

THEOREM A.1. The dependence analysis problem is undecidable.

We provide the briefest of outlines since the full proof runs into pages. We first show that for a class of grammars CG consisting of a set of context-free productions over the terminal symbols {0, 1, δ, T} along with the fixed set of non-context-free productions δ0 → e and 11 → e the problem of whether e belongs to an arbitrary grammar in the class is undecidable. This is done by reducing the halting problem to the to the e-recognition problem above. We then consider a subset of CG, say CG′, that is large enough to replay the undecidability proof. Specifically, the set CG′ represents encoding of all Turing Machines. Finally, we show that any grammar G in CG′ can be converted to a program P such that the problem of whether e belongs to L(G) can be reduced to the dependence analysis problem.

A.2 Soundness of Approximate Dependence Analysis

Recall the formal statement:

**THEOREM A.2.** If the dependence analysis of a program P with a demand σ main results in a demand environment in which the demand on an expression e is σ. Also consider a DGS trace of P with the same demand σ main on the main expression. For any evaluation of e in the trace, if the demand on e is δ, then δ ⊆ σ.

Consider the trace of a program in execution under DGS. Assume that an expression e appears on the trace for evaluation with an execution context E = (p, S, d, δ). The evaluation of e under the context E is deemed to be over, when its value v reaches the extent of evaluation specified by d and is replaced by the continuation on the top of δ. During this evaluation (of e under the context E), consider a sub-expression e′ of e that appears on the trace for evaluation with a context, say E′ = (p′, S′, d′, δ′). Then the soundness of our analysis involves showing that if σ and σ′ are the static demands on e and e′ respectively, then δ ⊆ σ implies δ′ ⊆ σ′. If this happens for every execution context E and every sub-expression e′, we say that the expression e preserves subsumption.

Assuming that all applications preserve subsumption, we first prove by structural induction that expressions preserve subsumption. Next we discharge the assumption regarding applications by showing that they too preserve subsumption. The only non-trivial application is a function call for which we induct on the depth of a call. Finally, starting with the fact that for σ main, δ main ⊆ σ main (actually δ main = σ main), we propagate the containment relation throughout the program using the previously proven subsumption results.

Detailed proofs of Theorem A.1 and Theorem A.2 can be found in [4].

A.3 Correctness of Incremental Slicing

We provide the detailed proof of the correctness of incremental slicing. Recall that we use the following notations:

(1) G′ is the grammar generated by dependence analysis for an expression π: e in the program of interest, when the slicing criteria is σ
(2) A is the automaton corresponding to G′ after Mohri-Nederhof transformation and canonicalization
(3) Aπ is the completing automaton for e

We first show that the result of the dependence analysis for an arbitrary slicing criterion σ can be decomposed as the concatenation of the grammar obtained from the dependence analysis with the fixed slicing criterion {e} and σ itself.

**LEMMA A.3.** For all expressions e and slicing criteria σ, L(G′) = L(G′) + L({e}).

Proof. The proof is by induction on the structure of e. Observe that all the rules of the dependence analysis (Figure 4) add symbols only as prefixes to the incoming demand. Hence, the slicing criteria will always appear as a suffix of any string that is produced by the grammar. Thus, any grammar L(G′) can be decomposed as σ′ ⊆ σ for some language σ′. Substituting {e} for σ, we get G′(e) = σ′. Thus L(G′) = L(G′) + L({e}).

Given a string s over (0 + 1)*, we use the notation s for the reverse of s in which all occurrences of 0 are replaced by 1 and 1 replaced by 0. Clearly, SL(s) = (e).

We next prove the completeness and minimality of Aπ.

**LEMMA A.4.** (s | SL(M′(s)) ∪ 0) = (Aπ(0 + 1)*).

Proof. We first prove bit by bit. Let a string s ∈ SL(M′(s)). Then by Lemma A.3, s ∈ SL(M′(s)). By Property 2, this also means that s ∈ SL(M′(s)). By Property 2, this also means that s ∈ SL(M′(s)). By Property 1, this means that there is a string s = s1s2 such that s1 ∈ (0 + 1 + 0)* and s2 ∈ (0 + 1)*, and SL((ps2)p1) ⊆ (0 + 1)*. Thus s can be split into two strings s1 and s2 such that SL((ps2)p1) = {e}. Therefore s1 = ps2. From the construction of Aπ, we have ps2 ∈ L(Aπ) and s1 ∈ (0 + 1)*. Therefore, s ∈ L(Aπ)(0 + 1)*.

Conversely, for the proof of RHS ⊆ LHS, we assume that a string s ∈ L(Aπ)(0 + 1)*. From the construction of Aπ, we have strings p1, p2, s′ such that p1p2 ∈ L(M′(s)) and p1 ∈ (0 + 1 + 0)* and p2 ∈ (0 + 1)*, s′ = p2s′. Thus, SL((ps2)p1) = SL(M′(s)) = SL(M′(s)) = SL(M′(s)). Thus, SL(M′(s)) is non-empty and s ∈ LHS.

We now prove our main result: Our slicing algorithm represented by inSlice (Algorithm 1) returns true if and only if SL(M′(s)) is non-empty.

**THEOREM A.5.** The incremental slicing algorithm is sound i.e. SL(M′(s)) = 0 ↔ inSlice(e, σ)

Proof. We first prove the forward implication. Let s ∈ SL(M′(s)). From Lemma A.3, s ∈ SL(M′(s)). From Property 2, s ∈ SL(M′(s)). Thus, there are strings p1, p2 such that p1 ∈ C(L(M′(s)), p2 ∈ σ, s = (p1p2). Further p1 in turn can be decomposed as p3p1 such that p3 ∈ (0 + 1 + 0)* and p1 ∈ (0 + 1)*. We also have SL((ps2)p1) ⊆ (0 + 1)*. Thus ps2 is a prefix of p2. Therefore, s belongs to SL(M′(s)).
From the construction of $\overrightarrow{A_π}$, we know $\overrightarrow{p_4} \in L(\overrightarrow{A_π})$. Further, $\overrightarrow{p_4}$ is a prefix of $p_2$ and $p_2 \in σ$, from the prefix closed property of $σ$ we have $\overrightarrow{p_4} \in σ$. This implies $\overrightarrow{A_π} \cap σ \neq \emptyset$ and thus inSlice($e, σ$) returns true.

Conversely, if inSlice($e, σ$) is true, then $\exists s$ : $s \in L(\overrightarrow{A_π}) \cap σ$. In particular, $s \in L(\overrightarrow{A_π})$. Thus, from Lemma A.4 we have $S(L(M_{s}^{π})) \neq \emptyset$. Further, since $s \in σ$ we have $S(L(M_{σ}^{π})) \neq \emptyset$. □

REFERENCES