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

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

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

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

ACM Reference Format:
Prasanna Kumar K., Amitabha Sanyal, Amey Karkare, and Saswat Padhi. 2019. A Static Slicing Method for Functional Programs and Its Incremental Version. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307345

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 (eg. lists) requires that the analysis should precisely capture structure transmitted data dependence. For example, it should be able to conclude that the value of (**car** (**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 **car** and **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

```
(define (lcc str lc cc)
  (if (null? str)
      (return (cons lc cc))
      (if (eq? (car str) nl)
          (return (lcc (cdr str) (+ lc 1) (+ cc 1)))
          (return (lcc (cdr str) lc (+ cc 1))))))
 (define (main)
   (return (lcc ... 0 0))))
```

Figure 1: A program in Scheme-like language to compute the number of characters and lines in the input string. Expressions underlined with a straight line are sliced away when the program is sliced with respect to car part of output and those underlined with a curvy line are sliced away when the program is sliced with respect to cdr.

do not contribute to the demand on the output. Such expressions are marked by an empty propagated demand. In this our method resembles the projection function based methods of [6] and [16]. However, unlike these methods which are context-insensitive, we get around the problem of undecidability by a suitable weakening of the requirement of context sensitivity. This makes our method precise by avoiding analysis over fewer infeasible interprocedural paths. To avoid the inefficiency of analyzing a function separately for each calling context, we create a compact context-independent summary for each function. This summary is then used to step over function calls. Interestingly, it is this context independent summary that enables the incremental version.

In certain applications of slicing such as parallelization, the same program has to be sliced more than once, each time with a different slicing criterion. The example from [16], reproduced in Figure 1, shows a program that takes a string as input and returns a pair consisting of the number of characters and lines in the string. To obtain a program in which character count and line count are computed in parallel, we could slice the program with respect to the **car** and the **cdr** of the result. This gives two versions of the program, one in which the expressions underlined with straight lines and the other in which the expressions underlined with curly lines are removed. After transforming these programs to make them executable, they can be run in parallel.

We extend the basic slicing technique to an incremental version that consists of a one-time pre-computation step that slices a program with respect to a *default criterion* that is common for all programs. The result of this step is converted to a set of automata, one for each expression in the program. Each automaton essentially represents the entire set of slicing criteria that keeps the corresponding expression in the slice. Interestingly, the set of automata can be computed in time that is comparable to the time for a single round of non-incremental slicing. 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.

$$
\begin{array}{rcll}
p \in \mathit{Prog} & ::= & d_1 \ldots d_n \;\; e_{\mathbf{main}} & \textit{— program} \\
df \in \mathit{Fdef} & ::= & (\mathbf{define}\ (f\ x_1\ \ldots\ x_n)\ e) & \textit{— function def} \\
e \in \mathit{Expr} & ::= & \left\{\begin{array}{ll} (\mathbf{if}\ x\ e_1\ e_2) & \textit{— conditional} \\ (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e) & \textit{— let binding} \\ (\mathbf{return}\ x) & \textit{— return from function} \end{array}\right. & \\
s \in \mathit{App} & ::= & \left\{\begin{array}{ll} k & \textit{— constant (numeric or } \mathbf{nil}\textit{)} \\ (\mathbf{cons}\ x_1\ x_2) & \textit{— constructor} \\ (\mathbf{car}\ x) & \textit{— selects first part of } \mathbf{cons} \\ (\mathbf{cdr}\ x) & \textit{— selects second part of } \mathbf{cons} \\ (\mathbf{null?}\ x) & \textit{— returns true if } x \textit{ is } \mathbf{nil} \\ (+\ x_1\ x_2) & \textit{— generic arithmetic} \\ (f\ x_1\ \ldots\ x_n) & \textit{— function application} \end{array}\right. &
\end{array}
$$

Figure 2: The syntax of our language

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 π 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 $e_{\mathbf{main}}$. Applications (denoted by the syntactic category *App*) consist of functions or operators applied to variables. Constants are regarded as 0-ary functions. Expressions (*Expr*) are either an **if** expression, a **let** expression that evaluates an application and binds the result to a variable, or a **return** expression. The **return** keyword is used to mark the end of a function.

3 DEPENDENCE ANALYSIS

We represent the selector **car** by **0** and **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 (**cons** (**cons** 1 2) 3), the substructure v' corresponding to (**cons** 1 2) with respect to the root of v can be represented by the set of access paths $\{\mathbf{0}, \mathbf{00}, \mathbf{01}\}$—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 α and demands by σ. Given two access paths α_1 and α_2, we use the juxtaposition $\alpha_1\alpha_2$ 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 $\{\alpha_1\alpha_2 \mid \alpha_1 \in \sigma \text{ and } \alpha_2 \in \sigma_2\}$ and $\mathbf{0}\sigma$ is a shorthand for $\{\mathbf{0}\alpha \mid \alpha \in \sigma\}$. We use the empty sequence of selectors ϵ to stand for the root of a structure. Thus base values such as integers have ϵ 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 σ representing the parts of interest in the value of e, and any other expression e', what parts σ' of the value of e' would be required to compute σ 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 $e_{\mathbf{main}}$ with a demand denoted by $\sigma_{\mathbf{main}}$.

As an example, consider the demand $\{\mathbf{10}\}$ on the expression (**cons** x y). Clearly, the computation of $\{\mathbf{10}\}$ part of (**cons** x y), requires $\{\mathbf{0}\}$ of y. We can thus think of (**cons** x y) as a *demand transformer* transforming the demand $\{\mathbf{10}\}$ on itself to demands on its subexpressions—$\{\mathbf{0}\}$ 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 $\delta_{\mathbf{main}}$ on the main expression of a program P, DGS computes the demand δ on each expression e, as it comes up for evaluation. δ represents extent to which e has to be evaluated to meet the required demand $\delta_{\mathbf{main}}$ on the main expression. We call the demand thus propagated to an expression as *dynamic demand* and denote it as δ to differentiate it from static demand σ. 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 $\delta_{\mathbf{main}}$. 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:

$$\begin{array}{llll} v : Val & = \mathbb{N} + \{\mathbf{nil}\} + Data \times Data & \text{– Evaluated} \\ d : Data & = Val + Clo & \text{– Data Value} \\ c : Clo & = (App \times Env) & \text{– Closure} \\ \rho : Env & = Var \to Data & \text{– Environment} \end{array}$$

An evaluated value is either a number or an empty list or a **cons** cell. A data value d may either be an evaluated value or a closure $\langle s, \rho \rangle$ in which s is an application, and ρ maps free variables of s to data values. An environment is a mapping from the set of variables *Var* to *Data*. The notation $[\vec{y} \mapsto \rho(\vec{x})]$ represents an environment that maps the formal arguments y_i to the bindings of the actual arguments x_i. $\rho \oplus \rho'$ represents the environment ρ shadowed by ρ' and $\lfloor \rho \rfloor_X$ represents the environment ρ' 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 $\rho, S, e, \delta \rightsquigarrow \rho', S', e', \delta'$. Here S is a stack of continuation frames, e is the current expression being evaluated, and δ is the dynamic demand on e. Each continuation frame is a 4-tuple $(\rho, x, e_{next}, \delta)$, signifying that the variable that is bound to the expression being evaluated is x, and e_{next} is the next expression to be evaluated in an environment ρ and with demand δ. The initial state of the transition system is $([\ ]_\rho, [([\ ]_\rho, \mathsf{ans}, (\mathbf{print}), \delta_{\mathbf{main}})], e_{\mathbf{main}}, \{\epsilon\})$. Notice the empty initial environment $[\ ]_\rho$, and the initial stack that has a single continuation frame. In this frame, ans is a distinguished variable that will eventually be bound to the value of $e_{\mathbf{main}}$ and (**print**) will be picked next for execution. The evaluation of $e_{\mathbf{main}}$ to the extent $\delta_{\mathbf{main}}$ is mediated through the **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 $\{\epsilon\}$), and passes the result to **print**. Evaluation stops if the extent of evaluation of the expression already satisfies its dynamic demand. Else **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 **let** and function calls follow rules of lazy evaluation. Closures are created by **let** expressions (LET) and evaluated at two places in a function body—while checking an **if** condition (IF-CLO) and at a **return** (RETURN-CLO). As an example of closure evaluation, we explain the rules for **car** and **cons**. If the demand δ on (**car** x) is $\emptyset$ then it is not evaluated at all (NO-EVAL). However, if δ is non-empty, the context of (**car** x) first does a **car**-selection on the value of x and then traverse the paths given by δ. This gives the demand on x as $\{\mathbf{0}\delta\}$.

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

Premise	Transition	Rule name
δ is $\emptyset$	$\rho, (\rho', y, e', \delta'):S, e, \delta \rightsquigarrow \rho', S, e', \delta'$	NO-EVAL
	$\rho, (\rho', y, e, \delta'):S, \kappa, \delta \rightsquigarrow \rho' \oplus \{y \mapsto \kappa\}, S, e, \delta'$	CONST
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, x, \delta \rightsquigarrow \rho', S, s, \delta$	VAR
	$\rho, S, (\mathbf{car}\ x), \delta \rightsquigarrow \rho, S, x, \mathbf{0}\delta$	CAR
	$\rho, S, (\mathbf{cdr}\ x), \delta \rightsquigarrow \rho, S, x, \mathbf{1}\delta$	CDR
	$\rho, (\rho', w, e, \delta'):S,\ (\mathbf{cons}\ x\ y), \{\epsilon\} \rightsquigarrow$ $\rho' \oplus \{w \mapsto (\rho(x), \rho(y))\}, S,\ e,\ \delta'$	CONS
$\delta' = \{\alpha \mid \mathbf{0}\alpha \in \delta\}$	$\rho, S,\ (\mathbf{cons}\ x\ y), \delta \rightsquigarrow \rho,\ S,\ x,\ \delta'$	CAR-CONS
$\delta' = \{\alpha \mid \mathbf{1}\alpha \in \delta\}$	$\rho, S,\ (\mathbf{cons}\ x\ y), \delta \rightsquigarrow \rho,\ S,\ y,\ \delta'$	CDR-CONS
$\rho(x), \rho(y) \in \mathbb{N}$	$\rho, (\rho', z, e, \delta'):S, (+\ x\ y), \delta \rightsquigarrow$ $\rho' \oplus \{z \mapsto (+\ \rho(x)\ \rho(y))\}, S, e, \delta'$	PRIM-ADD
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (+\ x\ y), \delta \rightsquigarrow \rho, (\rho, x, (+\ x\ y), \delta):S, x, \{\epsilon\}$	PRIM-1-CLO
$\rho(y)$ is $\langle s, \rho' \rangle$	$\rho, S, (+\ x\ y), \delta \rightsquigarrow \rho, (\rho, y, (+\ x\ y), \delta):S, y, \{\epsilon\}$	PRIM-2-CLO
f defined as $(\mathbf{define}\ (f\ \vec{y})\ e_f)$	$\rho, S, (f\ \vec{x}), \delta \rightsquigarrow [\vec{y} \mapsto \overrightarrow{\rho(x)}], S, e_f, \delta$	FUNCALL
	$\rho, S, (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e), \delta \rightsquigarrow \rho \oplus \{x \mapsto \langle s, \rho \rangle\}, S, e, \delta$	LET
$\rho(x) \in \mathbb{N}\ \&\ \rho(x) \neq 0$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, e_1, \delta$	IF-TRUE
$\rho(x) \in \mathbb{N}\ \&\ \rho(x) = 0$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, S, e_2, \delta$	IF-FALSE
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (\mathbf{if}\ x\ e_1\ e_2), \delta \rightsquigarrow \rho, (\rho, x, (\mathbf{if}\ x\ e_1\ e_2), \delta):S, x, \{\epsilon\}$	IF-CLO
$\rho(x)$ is $\langle s, \rho' \rangle$	$\rho, S, (\mathbf{return}\ x), \delta \rightsquigarrow \rho,\ S, x, \delta$	RETURN-CLO

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* $e_{\mathbf{main}}$ *as the main expression, a demand* $\delta_{\mathbf{main}}$*, a control point* π *and a string* $w \in (\mathbf{0} + \mathbf{1})^*$*, outputs yes if there exists a DGS trace of* $e_{\mathbf{main}}$ *with* $\delta_{\mathbf{main}}$ *in which the expression at* π *appears with a dynamic demand* δ *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 σ, $\mathcal{A}$ returns a demand environment that maps arguments of s to their demands. The third parameter to $\mathcal{A}$, denoted $\mathbb{DS}$, 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 ϵ 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 ϵ on both x and y.

Just as $\mathcal{A}$ 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 DEMAND-SUMMARY and results in a set of functions $\mathbb{DS}^i_f$, defining how a demand σ on a call to f is propagated to its ith parameter. The rule for function calls uses $\mathbb{DS}$ to propagate demands to the arguments of a specific call. We look upon the functions for $\mathbb{DS}_f$ as a context-independent summary of f—context-independent because it is parameterized with respect to the demand that will be instantiated at the place where the function is called.

The rule DEMAND-SUMMARY specifies the fixed-point property to be satisfied by $\mathbb{DS}$, 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 $\mathbb{DS}^i_f(\sigma)$ proceeds by analyzing e_f, the body of f, with respect to a *symbolic demand* σ. Then $\mathbb{DS}^i_f(\sigma)$ is the union of the demands on all occurrences of the ith argument in e_f. A call to a function, say g, in e_f is analysed using the summary $\mathbb{DS}_g$. In general, this results in a recursive description of $\mathbb{DS}^i_f(\sigma)$. We explain in Section 4 how to convert this to a closed form.

We finally discuss the rules for expressions given by $\mathcal{D}$. 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

$$\boxed{\mathcal{A} :: (\text{App}, \text{Demand}, \text{FuncSummaries}) \to \text{DemandEnvironment}}$$

$$\begin{array}{rcl}
\mathcal{A}(\pi{:}\kappa, \sigma, \mathbb{DS}) & = & \{\pi \mapsto \sigma\},\ \text{for constants including } \mathbf{nil} \\
\mathcal{A}(\pi{:}(\mathbf{null?}\ \pi_1{:}x), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}(+\ \pi_1{:}x\ \pi_2{:}y), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset, \\
 & & \quad \pi_2 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset\} \\
\mathcal{A}(\pi{:}\ (\mathbf{car}\ \pi_1{:}\ x), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \mathbf{0}\sigma\} \\
\mathcal{A}(\pi{:}\ (\mathbf{cdr}\ \pi_1{:}\ x), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \mathbf{1}\sigma\} \\
\mathcal{A}(\pi{:}(\mathbf{cons}\ \pi_1{:}x\ \pi_2{:}y), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \{\alpha \mid \mathbf{0}\alpha \in \sigma\}, \pi_2 \mapsto \{\beta \mid \mathbf{1}\beta \in \sigma\}\} \\
\mathcal{A}(\pi{:}(f\ \pi_1{:}y_1\ \cdots\ \pi_n{:}y_n), \sigma, \mathbb{DS}) & = & \bigcup_{i=1}^{n}\{\pi_i \mapsto \mathbb{DS}_f^i(\sigma)\}
\end{array}$$

$$\boxed{\mathcal{D} :: (\text{Exp}, \text{Demand}, \text{FuncSummaries}) \to \text{DemandEnvironment}}$$

$$\begin{array}{rcl}
\mathcal{D}(\pi{:}(\mathbf{return}\ \pi_1{:}x), \sigma, \mathbb{DS}) & = & \{\pi_1 \mapsto \sigma,\ \pi \mapsto \sigma\} \\
\mathcal{D}(\pi{:}(\mathbf{if}\ \pi_1{:}x\ e_1\ e_2), \sigma, \mathbb{DS}) & = & \mathcal{D}(e_1, \sigma, \mathbb{DS}) \cup \mathcal{D}(e_2, \sigma, \mathbb{DS}) \cup \\
 & & \{\pi_1 \mapsto \text{if } \sigma \neq \emptyset \text{ then } \{\epsilon\} \text{ else } \emptyset,\ \pi \mapsto \sigma\} \\
\mathcal{D}(\pi{:}(\mathbf{let}\ x\ \leftarrow\ \pi_1{:}s\ \mathbf{in}\ e), \sigma, \mathbb{DS}) & = & \mathcal{A}(s, \sigma', \mathbb{DS})\ \cup\ \{\pi \mapsto \sigma, \pi_1 \mapsto \sigma'\} \\
 & & \text{where } \Pi \text{ is the set of program points} \\
 & & \text{representing all occurrences of } x \text{ in } e \\
 & & \text{DE} = \mathcal{D}(e, \sigma, \mathbb{DS}),\ \text{and } \sigma' = \cup_{\pi' \in \Pi}\,\text{DE}(\pi'),
\end{array}$$

$$\boxed{\mathbb{DS} \in \text{FuncSummaries} :: \text{Funcname} \to (\text{Demand} \to (\text{Demand}_1, \ldots, \text{Demand}_n))}$$

$$\frac{\forall f, \forall i, \forall \sigma:\ \ \mathcal{D}(e_f, \sigma, \mathbb{DS}) = \text{DE}, \mathbb{DS}_f^i = \bigcup_{\pi \in \Pi}\ \text{DE}(\pi)}{df_1 \ldots df_k\ \vdash^l\ \mathbb{DS}}\ (\textsc{demand-summary})$$

$$\begin{array}{c}
(\mathbf{define}\ (f\ z_1\ \ldots\ z_n)\ \ e_f) \text{ is one of } df_1 \ldots df_k,\ 1 \leq i \leq n, \\
\text{and } \Pi \text{ represents all occurrences of } z_i \text{ in } e_f
\end{array}$$

Figure 4: Dependence Analysis

variable used for the condition gets an ϵ 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 σ 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 $e_{\mathbf{main}}$ is computed with the externally supplied demand $\sigma_{\mathbf{main}}$. Further, the demand on a function f, denoted σ_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 σ_f. If there is a call to g inside e_f, the demand summary $\mathbb{DS}_g$ 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 **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.

```
1.  (define (mmp xs p nv np xv xp)
2.    (if (null? xs)
3.        (return (cons (cons nv np) (cons xv xp)))
4.        (let p1← (+ π : p 1)
5.            (if (< (car xs) nv)
6.                (mmp (cdr xs) p1 (car xs) p xv xp))
7.                (if (> (car xs) xv)
8.                    (mmp (cdr xs) p1 nv np (car xs) p)
9.                    (mmp (cdr xs) p1 nv np xv xp)))))

10. (define (main)
11.   (return (mmp (cdr xs) 2 (car xs) 1 (car xs) 1)))
```

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.

Consider $\mathbb{DS}^2_{\mathbf{mmp}}$, the function that propagates the demand on a call to **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 **mmp** with a symbolic demand σ 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 σ 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 **mmp** at line 6 and the sixth in the call to

mmp at line 8. Clearly the demands on these two occurrences of p are $\mathbb{DS}^4_{\mathbf{mmp}}(\sigma)$ and $\mathbb{DS}^6_{\mathbf{mmp}}(\sigma)$. Also notice that the demands of the three occurrences of p1 at lines 6, 8 and 9 are the same, namely $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$. And since p is being used to define p1 at line 4, by the **let** rule, the demand on this occurrence of p is:

$$\text{if } (\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) \neq \emptyset) \text{ then } \{\epsilon\} \text{ else } \emptyset$$

Bringing everything together, we get:

$$\begin{aligned}\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) \quad = \quad & \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \cup \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \cup \\ & \text{if } (\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) \neq \emptyset) \text{ then } \{\epsilon\} \text{ else } \emptyset \end{aligned} \tag{1}$$

We bring this equation to a closed-form by substituting the values of $\mathbb{DS}^4_{\mathbf{mmp}}(\sigma)$ and $\mathbb{DS}^6_{\mathbf{mmp}}(\sigma)$ and eliminating the recursion in $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$. This is the subject of the next section. Also recollect that the concrete demand $\sigma_{\mathbf{mmp}}$ 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 $\sigma_{\mathbf{mmp}}$, 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 $\sigma_{\mathbf{main}}$. Thus we get:

$$\sigma_{\mathbf{mmp}} \quad = \quad \sigma_{\mathbf{mmp}} \cup \sigma_{\mathbf{main}} \tag{2}$$

This gives the value of $\sigma_{\mathbf{mmp}}$ as $\sigma_{\mathbf{main}}$. 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 $\sigma_{\mathbf{mmp}}$, the demand on p1 is $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma_{\mathbf{mmp}})$. Thus, by the let rule, the demand on p at π, denoted D_π, is given as:

$$\mathsf{D}_\pi \quad = \quad \text{if } (\mathbb{DS}^2_{\mathbf{mmp}}(\sigma_{\mathbf{mmp}}) \neq \emptyset) \text{ then } \{\epsilon\} \text{ else } \emptyset \tag{3}$$

In summary, at the end of the analysis we shall have (i) A set of equations (e.g. 1) defining the function summaries $\mathbb{DS}^i_f$ for each argument of each function, (ii) an equation (e.g. 2) for the concrete demand σ_f on the body of each function f, and (iii) equations (e.g. 3) specifying the demand D_π 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 $\sigma_{\mathbf{main}}$ 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 $\sigma_{\mathbf{main}}$ on the main expression. For any evaluation of e in the trace, if the demand on e is δ, then $\delta \subseteq \sigma$.*

4 COMPUTING DEPENDENCES

We now describe how to (i) set up the equations for function summaries, $\mathbb{DS}^i_f$, 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 requires us to do operations that cannot be done symbolically. The **cons** rule, for example, is defined in terms of the set $\{\alpha \mid \mathbf{0}\alpha \in \sigma\}$. Clearly this requires us to know whether the strings in σ start with a $\mathbf{0}$. Similarly, the **if** rule requires to know whether σ is $\emptyset$. The way out is to treat these operations also symbolically. For this we introduce three new symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\emptyset_\epsilon$, to capture the intended operations. If $\mathbf{0}$ represents selection using **car**, $\bar{\mathbf{0}}$ is intended to represent its opposite—use by the constructor **cons** as its left argument. Thus $\bar{\mathbf{0}}\mathbf{0}$ (**cons**-**car** cancellation) reduces to the empty string ϵ. Similarly $\emptyset_\epsilon$ 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 $\mathcal{S}$.

$$\begin{aligned}
&\mathcal{S}(\{\epsilon\}) = \{\epsilon\} \qquad \mathcal{S}(\mathbf{0}\sigma) = \mathbf{0}\mathcal{S}(\sigma) \qquad \mathcal{S}(\mathbf{1}\sigma) = \mathbf{1}\mathcal{S}(\sigma)\\
&\mathcal{S}(\bar{\mathbf{0}}\sigma) = \{\alpha \mid \mathbf{0}\alpha \in \mathcal{S}(\sigma)\}\\
&\mathcal{S}(\bar{\mathbf{1}}\sigma) = \{\alpha \mid \mathbf{1}\alpha \in \mathcal{S}(\sigma)\}\\
&\mathcal{S}(\emptyset_\epsilon\sigma) = \begin{cases} \emptyset & \text{if } \mathcal{S}(\sigma) = \emptyset \\ \{\epsilon\} & \text{otherwise}\end{cases}\\
&\mathcal{S}(\sigma_1 \cup \sigma_2) = \mathcal{S}(\sigma_1) \cup \mathcal{S}(\sigma_2)
\end{aligned}$$

The symbol $\bar{\mathbf{0}}$ strips the leading $\mathbf{0}$ from the string following it, as required by the rule for **cons**. Similarly, $\emptyset_\epsilon$[1] examines the string following it and replaces it by $\emptyset$ or $\{\epsilon\}$. The rules for **cons** and **if** in terms of the new symbols are:

$$\begin{aligned}
&\mathcal{A}(\pi\colon(\mathbf{cons}\ \pi_1\colon x\ \pi_2\colon y),\ \sigma,\ \mathbb{DS}) = \{\pi_1 \mapsto \bar{\mathbf{0}}\sigma,\ \pi_2 \mapsto \bar{\mathbf{1}}\sigma\}\\
&\mathcal{D}(\pi\colon(\mathbf{if}\ \pi_1\colon x\ e_1\ e_2),\ \sigma,\ \mathbb{DS}) = \mathcal{D}(e_1, \sigma, \mathbb{DS}) \cup \mathcal{D}(e_2, \sigma, \mathbb{DS}) \cup\\
&\qquad\qquad\qquad \{\pi_1 \mapsto \emptyset_\epsilon\sigma,\ \pi \mapsto \sigma\}
\end{aligned}$$

The rules for + and **null**? 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 σ_{sc} is substituted as a *concrete* demand for the main expression $e_{\mathbf{main}}$, i.e., $\sigma_{\mathbf{main}} = \sigma_{sc}$, we eliminate the symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\emptyset_\epsilon$ using $\mathcal{S}$.

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 $\mathbb{DS}^i_f$, we start with a symbolic demand σ and compute the demand environment for e_f, the body of f. From this we calculate the demand on the ith 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 ith argument is equated to $\mathbb{DS}^i_f(\sigma)$. Since the body may contain other calls, the dependence analysis within e_f makes use of $\mathbb{DS}$ in turn. The equations shown below define $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$. Notice that the earlier equation for $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$ in Section 3.2 has been rewritten in terms of the symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\emptyset_\epsilon$.

$$\begin{aligned}
\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) \quad &= \quad \mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \cup \mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \cup \emptyset_\epsilon\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)\\
\mathbb{DS}^4_{\mathbf{mmp}}(\sigma) \quad &= \quad \bar{\mathbf{1}}\bar{\mathbf{0}}\sigma \cup \mathbb{DS}^4_{\mathbf{mmp}}(\sigma)\\
\mathbb{DS}^6_{\mathbf{mmp}}(\sigma) \quad &= \quad \bar{\mathbf{1}}\bar{\mathbf{1}}\sigma \cup \mathbb{DS}^6_{\mathbf{mmp}}(\sigma)
\end{aligned}$$

[1] the choice of this symbol is to remind the reader that its action depends on whether the demand following it is $\emptyset$ or not.

As noted in [16], the main difficulty in obtaining a convenient function summary is to find a closed-form for $\mathbb{DS}^2_{\mathbf{mmp}}(\sigma)$ 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 σ with symbols, we can write $\mathbb{DS}^i_f(\sigma)$ as $\mathsf{DS}^i_f\sigma$ (DS^i_f concatenated with σ), where DS^i_f is a set of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \emptyset_\epsilon\}$, and represents the effect of $\mathbb{DS}^i_f$ on σ. The modified equations after doing this substitution will be,

$$\begin{aligned}\mathsf{DS}^2_{\mathbf{mmp}}\sigma &= \mathsf{DS}^6_{\mathbf{mmp}}\sigma \cup \mathsf{DS}^4_{\mathbf{mmp}}\sigma \cup \emptyset_\epsilon \mathsf{DS}^2_{\mathbf{mmp}}\sigma \\ \mathsf{DS}^4_{\mathbf{mmp}}\sigma &= \bar{\mathbf{1}}\bar{\mathbf{0}}\sigma \cup \mathsf{DS}^4_{\mathbf{mmp}}\sigma \\ \mathsf{DS}^6_{\mathbf{mmp}}\sigma &= \bar{\mathbf{1}}\bar{\mathbf{1}}\sigma \cup \mathsf{DS}^6_{\mathbf{mmp}}\sigma\end{aligned}$$

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

$$\mathbb{DS}^2_{\mathbf{mmp}}(\sigma) = \mathsf{DS}^2_{\mathbf{mmp}}\sigma, \text{ where} \tag{4}$$

$$\mathsf{DS}^2_{\mathbf{mmp}} = \mathsf{DS}^6_{\mathbf{mmp}} \cup \mathsf{DS}^4_{\mathbf{mmp}} \cup \emptyset_\epsilon \mathsf{DS}^2_{\mathbf{mmp}} \tag{5}$$

Notice that the equations for $\mathsf{DS}^2_{\mathbf{mmp}}$ and $\sigma_{\mathbf{mmp}}$ are still recursive. However, these equations can also be viewed as a grammar with $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{1}}, \bar{\mathbf{0}}, \emptyset_\epsilon\}$ as terminal symbols and $\mathsf{DS}^2_{\mathbf{mmp}}$, D_π and $\sigma_{\mathbf{mmp}}$ as non-terminals. Thus finding the solution to the set of equations generated by the dependence analysis reduces to finding the language generated by the corresponding grammar. In fact the language generated by the grammar is the least solution of equations above. The least solution corresponds to our interest, the most precise slice[2].

The equations can now be re-written as: grammar rules:

$$\begin{aligned}\mathsf{D}_\pi &\to \emptyset_\epsilon \mathsf{DS}^2_{\mathbf{mmp}}\sigma_{\mathbf{mmp}} \\ \mathsf{DS}^2_{\mathbf{mmp}} &\to \mathsf{DS}^4_{\mathbf{mmp}} \mid \mathsf{DS}^6_{\mathbf{mmp}} \mid \emptyset_\epsilon\ \mathsf{DS}^2_{\mathbf{mmp}} \\ \mathsf{DS}^4_{\mathbf{mmp}} &\to \bar{\mathbf{1}}\bar{\mathbf{0}} \mid \mathsf{DS}^4_{\mathbf{mmp}} \\ \mathsf{DS}^6_{\mathbf{mmp}} &\to \bar{\mathbf{1}}\bar{\mathbf{1}} \mid \mathsf{DS}^6_{\mathbf{mmp}} \\ \sigma_{\mathbf{mmp}} &\to \sigma_{\mathbf{sc}}\end{aligned} \tag{6}$$

Thus the question whether the expression at π can be sliced for the slicing criterion $\sigma_{\mathbf{mmp}}$ is equivalent to asking whether the language $\mathcal{S}(\mathsf{L}(\mathsf{D}_\pi))$ is empty. In fact, the simplification process $\mathcal{S}$ itself can be captured by adding the following set of five unrestricted productions named *unrestricted* and adding the production $\mathsf{D}'_\pi \to \mathsf{D}_\pi\$$ to the grammar generated earlier.

$$\bar{\mathbf{0}}\mathbf{0} \to \epsilon \qquad \bar{\mathbf{1}}\mathbf{1} \to \epsilon$$
$$\emptyset_\epsilon\mathbf{0} \to \emptyset_\epsilon \qquad \emptyset_\epsilon\mathbf{1} \to \emptyset_\epsilon \qquad \emptyset_\epsilon\$ \to \epsilon$$

The set of five unrestricted productions shown are independent of the program being sliced and the slicing criterion. The symbol $ marks the end of a sentence and is required to capture the $\emptyset_\epsilon$ rule correctly.

We now generalize and state our formulation of the slicing problem: Assume that π is the program point associated with an expression e. Given a slicing criterion σ, let G^σ_π denote the grammar $(N, T, P^\sigma_\pi \cup \textit{unrestricted} \cup \{\mathsf{D}'_\pi \to \mathsf{D}_\pi\$\}, \mathsf{D}'_\pi)$. Here T is the set of terminals $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \emptyset_\epsilon, \$\}$, P^σ_π is the set of context-free productions defining D_π, the demand on e (as illustrated by the grammar rules 6). N contains the non-terminals of P^σ_π and additionally includes the special non-terminal D'_π. As mentioned earlier, given a slicing criterion σ, the question of whether the expression e can be sliced out of the containing program is equivalent to asking whether the language $\mathsf{L}(G^\sigma_\pi)$ is empty. This is where the undecidability of the problem manifests in our specific approach:

Theorem 4.1. *Given a program point π and slicing criterion σ, the problem whether $\mathsf{L}(G^\sigma_\pi)$ is empty is undecidable.*

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^σ_π consists of a CFG, say G, along with the non-CFG productions in *unrestricted*. Mohri-Nederhoff is applied only to G, giving a regular grammar, say R. The membership problem of $R \cup \textit{unrestricted}$ is (efficiently) decidable. The NFA corresponding to this strongly regular grammar is denoted as M^σ_π. The simplification rules can be applied on M^σ_π 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 $G \cup \textit{unrestricted}$ 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-Nederhof transformation. The automata in Figure 6a–c and 6d–f correspond to the two slicing criteria $\sigma_{\mathbf{mmp}} = \{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{01}\}$ and $\sigma_{\mathbf{mmp}} = \{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{1}, \mathbf{10}\}$ and illustrate the simplification of corresponding Mohri-Nederhof automata $M^{\sigma_{\mathbf{mmp}}}_\pi$. It can be seen that, when the slicing criterion is $\{\epsilon, \mathbf{0}, \mathbf{00}, \mathbf{1}, \mathbf{10}\}$, the language of D_π 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-Nederhof approximation and simplification has to be repeated. This is likely to be inefficient for large programs.

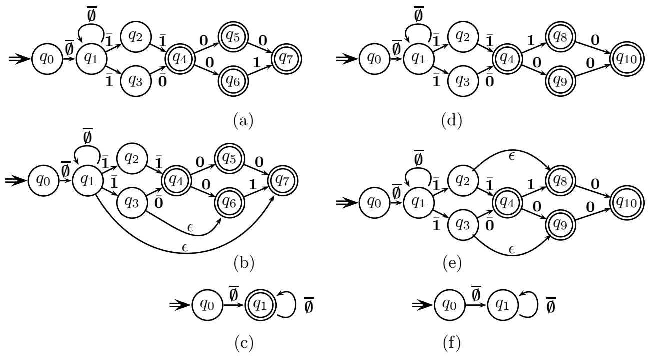

Figure 6: The simplification of the automaton M^σ_π: (a), (b) and (c) show the simplification for the slicing criterion $\sigma = \{\epsilon, \mathbf{0}, \mathbf{01}, \mathbf{00}\}$, while (d), (e) and (f) show the simplification for the criterion $\sigma = \{\epsilon, \mathbf{1}, \mathbf{0}, \mathbf{10}, \mathbf{00}\}$.

[2]We reiterate the differences between $\mathbb{DS}_f$ and DS_f. $\mathbb{DS}_f$ is a transfer function and is the unknown in the equation generated by the analysis. DS_f is a grammar symbol representing a set of strings, and is a part of the solution. The solution of $\mathbb{DS}_f$ maps a demand σ to $\mathsf{DS}_f\sigma$.

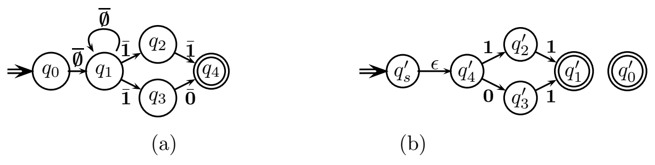

Figure 7: (a) The canonical automaton A_π and (b) the corresponding completing automaton $\overline{A}_\pi$

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 π: e and applying the Mohri-Nederhof procedure to construct the corresponding automaton $M_\pi^{\{\epsilon\}}$, (ii) a step called *canonicalization* which applies the simplification rules on $M_\pi^{\{\epsilon\}}$, but stops when the symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ 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 $M_\pi^{\{\epsilon\}}$ is shown in Figure 7a. Each accepting string in this automaton has $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols only at the end. Thus the automaton is canonical, and we shall denote it as A_π. It is clear that if A_π is concatenated with a slicing criterion that starts with the symbol $\mathbf{01}$, the result, after simplification, will be non-empty, and the expression at π has to be retained in the slice. We call such a string a *completing string* for A_π. Observe that the completing string was easy to detect since the canonicalization step pushed all the $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols towards the final state in the canonical automaton. Similarly, $\mathbf{11}$ is also a completing string for the same automaton.

Now consider the automaton in Figure 7b, called the *completing automaton*, that recognizes the language $(\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*$. This automaton recognizes *all* completing strings for A_π and nothing else. Thus for an arbitrary slicing criterion σ, it suffices to intersect σ with the completing automaton to decide whether the expression at π will be in the slice. In fact, it is enough for the completing automaton to recognize just the language $(\mathbf{01} + \mathbf{11})$ instead of $(\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*$. The reason is that any slicing criterion, say σ, is prefix closed, and therefore $\sigma \cap (\mathbf{01} + \mathbf{11})$ is $\emptyset$ if and only if $\sigma \cap (\mathbf{01} + \mathbf{11})(\mathbf{0} + \mathbf{1})^*$ 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 $M_e^{\{\epsilon\}}$ to an extent that all accepted strings have $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols only at the end. We now give a set of rules C, that captures this

```
Function createCompletingAutomaton($A$)
    Data: The Canonicalized Automaton
        $A = \langle Q, \{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \emptyset_\epsilon\}, \delta, q_0, F\rangle$
    Result: $\overline{A}$, the completing automaton for $A$
    $F' \leftarrow \{q_{\text{fr}} \mid q_{\text{fr}} \in Q,$ hasBarFreeTransition($q_0, q_{\text{fr}}, \delta$)$\}$
    /* Reverse the ‘‘bar’’ transitions: directions as
       well as labels                                        */
    foreach (transition $\delta(q, \bar{\mathbf{0}}) \rightarrow q'$) do
        add transition $\delta'(q', \mathbf{0}) \rightarrow q$
    foreach (transition $\delta(q, \bar{\mathbf{1}}) \rightarrow q'$) do
        add transition $\delta'(q', \mathbf{1}) \rightarrow q$
    $q'_s \leftarrow$ new state /* start state of $\overline{A}$            */
    foreach (state $q \in F$) do
        add transition $\delta'(q'_s, \epsilon) \rightarrow q$
    return $\langle Q \cup \{q'_s\}, \{\mathbf{0}, \mathbf{1}\}, \delta', q'_s, F'\rangle$
Function inSlice($e, \sigma$)
    Data: expression $e$, slicing criteria $\sigma$
    Result: Decides whether $e$ should be retained in slice
    return ($\mathsf{L}(\overline{A}_e) \cap \sigma \neq \emptyset$)
```

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

simplification.

$$C(\{\epsilon\}) = \{\epsilon\} \qquad C(\mathbf{0}\sigma) = \mathbf{0}C(\sigma)$$
$$C(\mathbf{1}\sigma) = \mathbf{1}C(\sigma) \qquad C(\emptyset_\epsilon\sigma) = \emptyset_\epsilon C(\sigma)$$
$$C(\bar{\mathbf{0}}\sigma) = \{\bar{\mathbf{0}} \mid C(\sigma) \text{ is } \{\epsilon\}\} \cup \{\alpha \mid \mathbf{0}\alpha \in C(\sigma)\}$$
$$\cup \{\bar{\mathbf{0}}\bar{\mathbf{1}}\alpha \mid \bar{\mathbf{1}}\alpha \in C(\sigma)\} \cup \{\bar{\mathbf{0}}\bar{\mathbf{0}}\alpha \mid \bar{\mathbf{0}}\alpha \in C(\sigma)\}$$
$$C(\bar{\mathbf{1}}\sigma) = \{\bar{\mathbf{1}} \mid C(\sigma) \text{ is } \{\epsilon\}\} \cup \{\alpha \mid \mathbf{1}\alpha \in C(\sigma)\}$$
$$\cup \{\bar{\mathbf{1}}\bar{\mathbf{1}}\alpha \mid \bar{\mathbf{1}}\alpha \in C(\sigma)\} \cup \{\bar{\mathbf{1}}\bar{\mathbf{0}}\alpha \mid \bar{\mathbf{0}}\alpha \in C(\sigma)\}$$
$$C(\sigma_1 \cup \sigma_2) = C(\sigma_1) \cup C(\sigma_2)$$

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

$$\mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathbf{00}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{1}}\bar{\mathbf{0}} \xrightarrow{C} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathbf{00}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{0}} \xrightarrow{C} \mathbf{1}\emptyset_\epsilon\mathbf{0}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\bar{\mathbf{0}}$$

In contrast the simplification using $\mathcal{S}$ gives:

$$\mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathbf{00}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{1}}\bar{\mathbf{0}} \xrightarrow{\mathcal{S}} \mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathbf{00}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\mathbf{1}\bar{\mathbf{1}}\emptyset \xrightarrow{\mathcal{S}}$$
$$\mathbf{1}\emptyset_\epsilon\bar{\mathbf{0}}\mathbf{00}\emptyset_\epsilon\mathbf{0}\bar{\mathbf{1}}\mathbf{0}\emptyset \xrightarrow{\mathcal{S}} \ldots \xrightarrow{\mathcal{S}} \emptyset$$

C satisfies two important properties:

(1) The result of C always has the form $(\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*(\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. Further, if $\sigma \subseteq (\mathbf{0} + \mathbf{1} + \emptyset_\epsilon)^*$, then $C(\sigma) = \sigma$.
(2) $\mathcal{S}$ subsumes C, i.e., $\mathcal{S}(C(\sigma_1)C(\sigma_2)) = \mathcal{S}(\sigma_1\sigma_2)$.

Note that while we have defined canonicalization over a language, the actual canonicalization takes place over an automaton—specifically the automaton M_π obtained after Mohri-Nederhof transformation. The process of canonicalization over an automaton is a minor variation of the simplification process [5]. Specifically, (i) adjacent $\bar{\mathbf{0}}\mathbf{0}$ and $\bar{\mathbf{1}}\mathbf{1}$ edges are replaced by an ϵ edge and the resulting automaton is made deterministic, until there are no more such edges, and (ii) edges with labels $\bar{\mathbf{0}}$ or $\bar{\mathbf{1}}$ are retained only if their targets have a path reaching some final node, and the labels on this path consist only of $\bar{\mathbf{0}}$ or $\bar{\mathbf{1}}$ symbols. It is in the second step that the canonicalization differs from simplification over automata.

Algorithm 1 describes function **createCompletingAutomaton** that takes A_π, the canonical Mohri-Nederhof automaton for the slicing criterion $\{\epsilon\}$, as input, and constructs the completing automaton $\overline{A}_\pi$. Recollect that the strings recognized by A_π are from $(\mathbf{0} + \mathbf{1} + \emptyset_\epsilon)^*(\bar{\mathbf{0}} + \bar{\mathbf{1}})^*$. Call the set of states reachable from the start state using only edges with labels $\{\mathbf{0}, \mathbf{1}, \emptyset_\epsilon\}$ as the *frontier set*. The completing automaton is a copy of canonical automaton with edges labeled by $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols reversed, and the symbols themselves replaced by $\mathbf{0}$ and $\mathbf{1}$ respectively. All edges with labels $\{\mathbf{0}, \mathbf{1}, \emptyset_\epsilon\}$ 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 π: e can be sliced out, the function **inSlice** described in Algorithm 1 just checks if the intersection of the slicing criteria with $\mathsf{L}(\overline{A}_\pi)$ is null. We now present a theorem that states that the incremental algorithm **inSlice**(e, σ) 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. $\mathcal{S}(\mathsf{L}(M^\sigma_\pi)) \neq \emptyset \leftrightarrow$ **inSlice**(e, σ)

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 creating the completing automata is amortized over only a few slicing criteria.

Our benchmarks consists of first order programs derived from the nofib 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 λ-expression evaluator that returns a tuple consisting of an atomic value and a list. The criterion $\{\epsilon, 0\}$ 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 $\{\epsilon\}$ or $\{\epsilon, 1\}$ 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: {**main**() -> x = {1,2}, {y,z} = x, y}, their method gives the imprecise slice {**main**() -> 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 (**let** x← (**cons** 1 2) **in** (**let** y ← (**car** x) **in** y)) with the slicing criterion ϵ, our method would correctly compute the demand on the constant 2 as $\bar{\mathbf{1}}(\epsilon \cup \mathbf{0})$. This simplifies to the demand $\emptyset$, 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 ψ, 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 π: (**cons** π_1: x π_2: y) representing the body of a function. Dependence analysis with the symbolic demand σ gives the demand environment $\{\pi \mapsto \sigma, \pi_1 \mapsto \bar{\mathbf{0}}\sigma, \pi_2 \mapsto \bar{\mathbf{1}}\sigma\}$. Notice that the demands π_1 and π_2 are in terms of the symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$, a result of our decision to work with symbolic demands. Slicing with the default criterion ϵ and canonicalizing the result gives the demand environment $\{\pi \mapsto \epsilon, \pi_1 \mapsto \bar{\mathbf{0}}, \pi_2 \mapsto \bar{\mathbf{1}}\}$. There is enough information in this to deduce that π_1 (π_2) will be in the slice only if the slicing criterion includes $\mathbf{0}$($\mathbf{1}$). 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 ψ, they calculate a program equivalent to $\psi \circ P$. However the method is not automated. Finally,

Table 1: Statistics for incremental and non-incremental slicing.

Program	Pre-computation time (ms)	#exprs in program	Slicing with $\{\epsilon\}$			Slicing with $\{\epsilon, 0\}$			Slicing with $\{\epsilon, 1\}$		
			Non-inc time (ms)	Inc time (ms)	#expr in slice	Non-inc time (ms)	Inc time (ms)	#expr in slice	Non-inc time (ms)	Inc time (ms)	#expr in slice
First-order Programs											
treejoin	1817.2	609	1556.7	1.3	565	1592.3	1.4	567	1641.3	1.6	567
deriv	217.0	415	109.6	0.9	267	128.3	0.9	273	173.5	1.3	293
minmaxpos	32.4	189	13.7	0.7	166	16.8	0.7	168	16.5	0.6	168
nperm	408.5	648	285.1	1.3	249	303.7	5.2	429	298.9	5.4	284
paraffins	1898.1	1268	1775.1	1.3	16	1791.8	4.0	1204	1803.1	4.0	1204
knightstour	834.1	675	647.9	1.4	511	691.3	3.2	511	546.8	2.9	511
huffman	4208.2	1124	3324.0	2.3	897	3559.5	2.2	900	3373.0	2.2	901
sudoku	94273.3	2105	80065.4	4.6	2070	83629.5	3.9	2073	79691.1	4.7	2070
lcss	10144.0	703	6773.0	1.5	694	7474.0	1.5	697	7714.8	1.6	695
nqueens	165.0	366	96.8	0.9	366	116.4	0.9	366	116.7	0.9	366
linecharcount	17.6	108	8.8	0.5	92	6.7	0.6	98	8.4	0.4	102
studentinfo	90.6	310	64.7	0.6	106	67.6	0.7	109	71.1	1.0	108
takl	43.7	158	25.3	0.4	105	27.0	0.4	111	26.5	0.6	105
fibheap	133716.3	679	101365.1	2.2	53	101508.9	10.7	678	102740.1	8.7	54
lambda	1625.9	750	1179.2	1.6	27	1335.7	5.7	734	1014.2	2.8	34
Higher-order Programs											
fold	25.0	127	11.2	0.3	17	15.0	0.5	84	12.9	0.5	36
barneshut	7859.1	1600	5876.2	2.8	149	6121.1	9.1	611	5989.9	3.2	151
maptail	20.3	102	13.8	0.6	58	11.9	0.6	72	12.9	0.7	64
sssp	2913.9	772	2260.9	1.3	385	2084.9	2.9	429	2296.4	2.0	385
parser	3711.4	1294	2473.1	1.1	178	2623.7	1.0	178	3160.3	1.3	184

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 ϵ. 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 undecidabilty. 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 **mapsq**:

```
(define (mapsq l)
  (if (null? l) (return l)
    (return (cons (sq (car l)) (mapsq (cdr l))))))
```

The reader can verify that the actual function summary for **mapsq** is: $\mathbb{DS}^1_{\mathbf{mapsq}}(\sigma) = \mathsf{DS}^1_{\mathbf{mapsq}}\sigma$, where $\mathsf{DS}^1_{\mathbf{mapsq}}$ is the language $\epsilon \mid \mathbf{1}^n\bar{\mathbf{1}}^n \mid \mathbf{1}^n\mathbf{0}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}^n$, for $n \geq 0$. Now, given a slicing criterion $\sigma = \{\epsilon, \mathbf{1}, \mathbf{11}, \mathbf{110}\}$ standing for the path to the third element of a list, it is easy to see that $\mathbb{DS}^1_{\mathbf{mapsq}}(\sigma)$ after simplification would give back σ itself, and this is the most precise slice. However, due to Mohri-Nederhof approximation $\mathsf{DS}^1_{\mathbf{mapsq}}$ would be approximated by $\epsilon \mid \mathbf{1}^n\bar{\mathbf{1}}^m \mid \mathbf{1}^k\mathbf{0}\emptyset_\epsilon\bar{\mathbf{0}}\bar{\mathbf{1}}^l$, $n, m, k, l \geq 0$. In this case, $\mathsf{DS}^1_{\mathbf{mapsq}}$ would be $(\mathbf{0}+\mathbf{1})^*$, keeping all the elements of the input list $\mathtt{l}$ in the slice.

A APPENDIX

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 $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$ along with the fixed set of non-context-free productions $\bar{\mathbf{0}}\mathbf{0} \to \epsilon$ and $\bar{\mathbf{1}}\mathbf{1} \to \epsilon$ the problem of whether ϵ belongs to an arbitrary grammar in the class is undecidable. This is done by reducing the halting problem to the to the ϵ-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 ϵ belongs to $\mathsf{L}(G)$ 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 $\sigma_{\mathbf{main}}$ 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 $\sigma_{\mathbf{main}}$ on the main expression. For any evaluation of e in the trace, if the demand on e is δ, then $\delta \subseteq \sigma$.*

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 $\mathcal{E} = (\rho, S, \_, \delta)$. The evaluation of e under the context $\mathcal{E}$ is deemed to be over, when its value v reaches the extent of evaluation specified by δ and is replaced by the continuation on the top of S. During this evaluation (of e under the context $\mathcal{E}$), consider a sub-expression e' of e that appears on the trace for evaluation with a context, say $\mathcal{E}' = (\rho', S', \_, \delta')$. Then the soundness of our analysis involves showing that if σ and σ' are the static demands on e and e' respectively, then $\delta \subseteq \sigma$ implies $\delta' \subseteq \sigma'$. If this happens for every execution context $\mathcal{E}$ 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 $e_{\mathbf{main}}$, $\delta_{\mathbf{main}} \subseteq \sigma_{\mathbf{main}}$ (actually $\delta_{\mathbf{main}} = \sigma_{\mathbf{main}}$), 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^σ_π is the grammar generated by dependence analysis for an expression $\pi\colon e$ in the program of interest, when the slicing criteria is σ
(2) A_π is the automaton corresponding to $G^{\{\epsilon\}}_\pi$ after Mohri-Nederhof transformation and canonicalization
(3) $\overline{A}_\pi$ 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 $\{\epsilon\}$ and σ itself.

Lemma A.3. *For all expressions e and slicing criteria σ, $\mathsf{L}(G^\sigma_\pi) = \mathsf{L}(G^{\{\epsilon\}}_\pi)\sigma$.*

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 $\mathsf{L}(G^\sigma_\pi)$ can be decomposed as $\sigma'\sigma$ for some language σ'. Substituting $\{\epsilon\}$ for σ, we get $G^{\{\epsilon\}}_\pi = \sigma'$. Thus $\mathsf{L}(G^\sigma_\pi) = \mathsf{L}(G^{\{\epsilon\}}_\pi)\sigma$. □

Given a string s over $(\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, we use the notation $\bar{s}$ to stand for the reverse of s in which all occurrences of $\bar{\mathbf{0}}$ are replaced by $\mathbf{0}$ and $\bar{\mathbf{1}}$ replaced by $\mathbf{1}$. Clearly, $\mathcal{S}(\{s\bar{s}\}) = \{\epsilon\}$.

We next prove the completeness and minimality of $\overline{A}_\pi$.

Lemma A.4. $\{s \mid \mathcal{S}(\mathsf{L}(M^{\{s\}}_\pi)) \neq \emptyset\} = \mathsf{L}(\overline{A}_\pi)(\mathbf{0}+\mathbf{1})^*$

Proof. We first prove $LHS \subseteq RHS$. Let the string $s \in \mathcal{S}(\mathsf{L}(M^{\{s\}}_\pi))$. Then by Lemma A.3, $s \in \mathcal{S}(\mathsf{L}(M^{\{\epsilon\}}_\pi)\{s\})$. By Property 2, this also means that $s \in \mathcal{S}(C(\mathsf{L}(M^{\{\epsilon\}}_\pi))\{s\})$. Since strings in $C(\mathsf{L}(M^{\{\epsilon\}}_\pi))$ are of the form $(\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*(\bar{\mathbf{0}}+\bar{\mathbf{1}}))^*$ (Property 1), this means that there is a string p_1p_2 such that $p_1 \in (\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*$ and $p_2 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, and $\mathcal{S}(\{p_2\}\{s\}) \subseteq (\mathbf{0}+\mathbf{1})^*$. Thus s can be split into two strings s_1 and s_2, such that $\mathcal{S}(\{p_2\}\{s_1\}) = \{\epsilon\}$. Therefore $s_1 = \overline{p_2}$. From the construction of $\overline{A}_\pi$ we have $\overline{p_2} \in \mathsf{L}(\overline{A}_\pi)$ and $s_2 \in (\mathbf{0}+\mathbf{1})^*$. Thus, $s \in \mathsf{L}(\overline{A}_\pi)(\mathbf{0}+\mathbf{1})^*$.

Conversely, for the proof of $RHS \subseteq LHS$, we assume that a string $s \in \mathsf{L}(\overline{A}_\pi)(\mathbf{0}+\mathbf{1})^*$. From the construction of $\overline{A}_\pi$ we have strings p_1, p_2, s' such that $p_1p_2 \in C(\mathsf{L}(M^\epsilon_\pi)), p_1 \in (\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*, p_2 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$, s is $\overline{p_2}s'$ and $s' \in (\mathbf{0}+\mathbf{1})^*$. Thus, $\mathcal{S}(\mathsf{L}(M^{\{s\}}_\pi)) = \mathcal{S}(\mathsf{L}(M^{\{\epsilon\}}_\pi\{s\})) = \mathcal{S}(C(\mathsf{L}(M^{\{\epsilon\}}_\pi))\{s\}) = \mathcal{S}(\{p_1p_2\overline{p_2}s'\}) = \{p_1s'\}$. Thus, $\mathcal{S}(\mathsf{L}(M^{\{s\}}_\pi))$ is non-empty and $s \in LHS$. □

We now prove our main result: Our slicing algorithm represented by **inSlice** (Algorithm 1) returns true if and only if $\mathcal{S}(\mathsf{L}(A^\epsilon_\pi)\sigma)$ is non-empty.

Theorem A.5. *The incremental slicing algorithm is sound i.e.* $\mathcal{S}(\mathsf{L}(M^\sigma_\pi)) \neq \emptyset \leftrightarrow$ **inSlice**(e, σ)

Proof. We first prove the forward implication. Let $s \in \mathcal{S}(\mathsf{L}(M^\sigma_\pi))$. From Lemma A.3, $s \in \mathcal{S}(\mathsf{L}(M^\epsilon_\pi)\sigma)$. From Property 2, $s \in \mathcal{S}(C(\mathsf{L}(M^\epsilon_\pi))\sigma)$. Thus, there are strings p_1, p_2 such that $p_1 \in C(\mathsf{L}(M^\epsilon_\pi))$, $p_2 \in \sigma, s = \mathcal{S}(\{p_1p_2\})$. Further p_1 in turn can be decomposed as p_3p_4 such that $p_3 \in (\mathbf{0}+\mathbf{1}+\emptyset_\epsilon)^*$ and $p_4 \in (\bar{\mathbf{0}}+\bar{\mathbf{1}})^*$. We also have $\mathcal{S}(\{p_4p_2\}) \subseteq (\mathbf{0}+\mathbf{1})^*$. Thus $\overline{p_4}$ is a prefix of p_2.

From the construction of $\overline{A}_\pi$, we know $\overline{p_4} \in \mathsf{L}(\overline{A}_\pi)$. Further, $\overline{p_4}$ is a prefix of p_2 and $p_2 \in \sigma$, from the prefix closed property of σ we have $\overline{p_4} \in \sigma$. This implies $\overline{A}_\pi \cap \sigma \neq \emptyset$ and thus **inSlice**(e, σ) returns true.

Conversely, if **inSlice**(e, σ) is true, then $\exists s : s \in \mathsf{L}(\overline{A}_\pi) \cap \sigma$. In particular, $s \in \mathsf{L}(\overline{A}_\pi)$. Thus, from Lemma A.4 we have $\mathcal{S}(\mathsf{L}(M_\pi^{\{s\}})) \neq \emptyset$. Further, since $s \in \sigma$ we have $\mathcal{S}(\mathsf{L}(M_\pi^{\sigma})) \neq \emptyset$. □

REFERENCES

[1] David Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Advances in Computers* 62 (2004).

[2] Sandip Kumar Biswas. 1997. *Dynamic Slicing in Higher-order Programming Languages*. Ph.D. Dissertation. University of Pennsylvania, Philadelphia, PA, USA.

[3] R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. http://www.jstor.org/stable/1995158

[4] Prasanna Kumar K. 2019. *Dependence Analysis of Functional Programs and its Applications*. Ph.D. Dissertation. Indian Institute of Technology, Bombay, Mumbai, India.

[5] Amey Karkare, Uday Khedker, and Amitabha Sanyal. 2007. Liveness of Heap Data for Functional Programs. In *Heap Analysis and Verification Workshop*.

[6] Yanhong A. Liu and Scott D. Stoller. 2003. Eliminating Dead Code on Recursive Data. *Sci. Comput. Program.* 47 (2003).

[7] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

[8] Neil Mitchell and Colin Runciman. 2009. Losing Functions Without Gaining Data: Another Look at Defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*.

[9] Mehryar Mohri and Mark-Jan Nederhof. 2000. Regular Approximation of Context-Free Grammars through Transformation. In *Robustness in Language and Speech Technology*. Kluwer Academic Publishers.

[10] NoFib. 2019. Haskell Benchmark Suite. http://git.haskell.org/nofib.git. (Last accessed).

[11] Claudio Ochoa, Josep Silva, and Germán Vidal. 2008. Dynamic Slicing of Lazy Functional Programs Based on Redex Trails. *Higher Order Symbol. Comput.* 21 (2008).

[12] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. In *ACM SIGPLAN International Conference on Functional Programming*.

[13] Simon L. Peyton-Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall.

[14] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[15] Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Trans. Program. Lang. Syst.* (2000).

[16] Thomas W. Reps and Todd Turnidge. 1996. Program Specialization via Program Slicing. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany*.

[17] Nuno F. Rodrigues and Luis S. Barbosa. 2006. Component Identification Through Program Slicing. *Electronic Notes in Theoretical Computer Science* 160 (2006).

[18] Nuno F. Rodrigues and Luis S. Barbosa. 2006. Program Slicing by Calculation. *Journal of Universal Computer Science* (2006).

[19] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-passing Style. *SIGPLAN Lisp Pointers* (1992).

[20] Josep Silva. 2012. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.* (2012).

[21] Josep Silva, Salvador Tamarit, and César Tomás. 2012. System Dependence Graphs in Sequential Erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12)*.

[22] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3 (1995).