논•문 16-3-2

Bi-directional Demand-Driven Set-Based Analysis

Woongsik Choi and Kwangkeun Yi
Korea Advanced Institute of Science and Technology
Division of Computer Science
{wschoi, kwang}@ropas.kaist.ac.kr

Abstract

In this paper, we present a bi-directional demand-driven set-based analysis which solves only those constraints affecting program points one wants to analyze. To achieve this goal, we incorporate the notion of demands on set constraints and analyze programs in two directions: forward to know which values flow into a given point, and backward to know which program points that a given value flows into. We prove that for interested program points, our analysis gives exactly the same results as whole-program set-based analysis. As an experiment of our approach, we analyzed each program point separately using our demand-driven formulation. We report the efficiency of our analysis as percentage of constraints solved for each program point compared with the whole-program analysis.

1. Introduction

When analyzing programs, there are often situations that analysis of whole program is not needed. For example, one may need analysis results for only a subset of all functions in the program. For array bound check, only values for array index are needed. Another situation might be the semantic exhaustiveness check of pattern matchings in ML-like programs, where one may need only those values flowing into pattern matchings.

With these motivations, we propose demand-driven formulation of set-based analysis. Set-based analysis [5,4,3] is a static analysis technique that estimates sets of values by regular tree grammars [4]. Such regular

tree grammar's production rules are derived in a form of set-constraints. In set-based analysis, there's no solving order defined and due to higher order functions in functional languages, analysis results for a program point can't be determined until analysis for whole-program is finished. To overcome this aspect and make it demand-driven, we incorporate notion of demands to set-based analysis. Starting from initial demands of interested program points, we add new demands for related program points and solve a constraint only if we have demand for it. One can view demands as specifying solving order to set-based analysis.

To achieve demand-driven formulation in the presence of higher-order functions, we solve constraints in two directions. For a given program point, the result of set-based analysis is approximation of all values computed for the point. To acquire all such "forward" information of computation in demand-driven way, we sometimes need opposite "backward" information of all program points on which a given function can be called. Let's consider following higher-order program fragment.

```
let fun f x = case x of ...
    fun g h = ... e1 ...
in ... f e2 ... g f ... end
```

For the (forward) demand of argument x, we have to collect function f's actual arguments, e_1 and e_2 by identifying all call sites of f. e_2 might be easily identified syntactically, but as functions in higher-order programs are first-class objects, e_1 can't be identified without analyzing all call sites the function f flows into. So, we add backward demand for f and solve this demand in backward direction identifying that f flows into the call site h e_1.

1.1. Other Approaches for Demand Driven Analyses

Biswas formulated a demand-driven set-based analysis for the purpose of dead code elimination [2]. However, in his formulation, initial demand is restricted only to top-level expression and constraints for arbitrary program points can't be analyzed separately. He adds demand constraints to usual value constraints. His approach solves both value constraints and demand constraints simultaneously. Demand constraints are solved in such a way that if a program point is included in the demand, then it is not dead code. In this way, he solves constraints for only non-dead codes, and when solving is done, program points not in the demand set are identified as dead codes. Contrary to this use of demands to indicate non-dead codes, demands in our demand-driven formulation can be any program points of interest.

In Heintze and Tardieu's demand-driven formulation of pointer analysis for C-like

Var	x, y	variables	
Con	c	data constructors	
$Expr$			
e	$::=$	x	variable
	$\mid$	$c(e_1, \ldots, e_n)$	data construction
	$\mid$	$\lambda x.e$	function abstraction
	$\mid$	$e_1\ e_2$	function application
	$\mid$	$case(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3)$	pattern matching

Figure 1. Syntax

languages[6], similar concept of separating forward flows and backward flows is used. In a language like C, pointer analysis is answering the query "what may a program variable p point to?". To solve indirect assignment in fully demand-driven way, they solve the backward query "what variables may point to p?". Our work can be seen as applying their strategies for demand-driven formulation to set-based analysis of higher-order languages.

1.2. Paper Organization

The rest of this paper is organized as follows. In Section 2, we define a core ML language we will use to represent our work. In Section 3, we briefly summarize set-based analysis for the language. In Section 4, we modify the set-based analysis and formulate a demand-driven set-based analysis. In Section 5, we prove that our demand-driven formulation gives the same results for demanded program points. In Section 6, we report the experimental results for solving each program points separately and discuss about the results. In Section 7, we conclude the paper.

2. The Language

In this section, we introduce a simple language that we will describe our work with. It is an untyped core ML with data constructors and simplified pattern matching constructs.

Syntax of the language is shown in Figure 1. Here, Var is a finite set of program variables and Con is a finite set of date constructors with predefined arities. x, $\lambda x.e$, $e_1\ e_2$, are components of core ML. $c(e_1, \cdots, e_n)$ produces structured values with data constructor c of fixed arity n. $case(e_1, c(x_1, \cdots, x_n) \Rightarrow e_2, y \Rightarrow e_3)$ represents simplified pattern matching. Recursive functions are omitted for presentational simplicity because they are not affected by our demand-driven formulation.

We show dynamic semantics of the language in natural semantics form in Figure 2. The natural semantics establish the relation $E \vdash e \rightarrow v$ saying that in environment E, expression e evaluates to value v. Here, environment E is a finite mapping from

$$\begin{array}{lll} Env\ E & \in & Var \xrightarrow{fin} Val \quad \text{environment} \\ Val\ v & ::= & c(v_1, \ldots, v_n) \quad \text{constructed value} \\ & | & \langle E, \lambda x.e \rangle \quad \text{function closure} \end{array}$$

$$\frac{}{E \vdash x \to E(x)}$$

$$\frac{E \vdash e_i \to v_i \quad 1 \le i \le n}{E \vdash c(e_1, \ldots, e_n) \to c(v_1, \ldots, v_n)}$$

$$\frac{}{E \vdash \lambda x.e \to \langle E, \lambda x.e \rangle}$$

$$\frac{E \vdash e_1 \to \langle E', \lambda x.e \rangle \quad E \vdash e_2 \to v' \quad E'[x \mapsto v'] \vdash e \to v}{E \vdash e_1\ e_2 \to v}$$

$$\frac{E \vdash e_1 \to c(v_1, \ldots, v_n) \quad E[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e_2 \to v}{E \vdash case(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v}$$

$$\frac{E \vdash e_1 \to v' \quad E[y \mapsto v'] \vdash e_3 \to v}{E \vdash case(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v} \quad v' \neq c(\ldots)$$

Figure 2. Dynamic semantics

program variables to values. The notation $E[x \mapsto v]$ denotes the environment which has the same mappings as E except for x, which maps to v. We have two kinds of values, data constructed values and function closures. To enforce static scoping, we keep track of the environment in which a function is generated as a function closure.

3. Set-Based Analysis [5]

Set-based analysis is a static analysis method, which approximates sets of values for each programs points by regular tree grammars [4], whose production rules are derived in a form of set constraints. By collapsing all environments appearing during evaluation into a single set environment, dependencies between different environments are ignored and runtime values of the program are approximated in finite time.

The method is separated into two phases. In first phase, initial set constraints are generated from the source program. This generation transforms the meaning of the program into set containment relationships with set expressions modeling approximation semantics. Also, every program point is named with set variables. So, after generation, the source program is not needed anymore. In second phase, we solve the generated constraints by adding simpler constraints explicitly representing flows of values. Analysis is done when there are no more constraints to add. The analysis result is explicit set containments for every set variable.

3.1. Set Constraints

$SetVar\ \mathcal{W}, \mathcal{X}, \mathcal{Y}$ set variables
$SetExp$ set expression

$se ::= \mathcal{W} \mid \mathcal{X} \mid \mathcal{Y}$		variable
	$\mid \lambda x.e$	function value
	$\mid c(\mathcal{X}_1, \ldots, \mathcal{X}_n)$	constructed value
	$\mid apply(\mathcal{X}_1, \mathcal{X}_2)$	application
	$\mid case(\mathcal{Y}_1, c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)$	pattern matching

Atomic set expression
$ae ::= \lambda x.e \mid c(\mathcal{X}_1, \ldots, \mathcal{X}_n)$

Special set variables
- $dom(\lambda x.e)$: formal argument (x)
- $ran(\lambda x.e)$: body expression (e)

Meaning of set expressions

$Val\ \ v ::= c(v_1, \ldots, v_n)$	constructed value
$\mid \lambda x.e$	function value
$\mathcal{I} \in SetVar \rightarrow 2^{Val}$	interpretation

Extension of $\mathcal{I}$
- $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$
- $\mathcal{I}(c(\mathcal{X}_1, \ldots, \mathcal{X}_n)) = \{c(v_1, \ldots, v_n) \mid v_i \in \mathcal{I}(\mathcal{X}_i),\ 1 \le i \le n\}$
- $\mathcal{I}(apply(\mathcal{X}_1, \mathcal{X}_2)) = \{v \mid \lambda x.e \in \mathcal{I}(\mathcal{X}_1),\ v \in ran(\lambda x.e)\}$
  provided $\lambda x.e \in \mathcal{I}(\mathcal{X}_1)$ implies $\mathcal{I}(\mathcal{X}_2) \subseteq \mathcal{I}(dom(\lambda x.e))$
- $\mathcal{I}(case(\mathcal{Y}_1,\ c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2,\ \mathcal{W} \Rightarrow \mathcal{Y}_3)) =$
  $\{v \mid v \in \mathcal{I}(\mathcal{Y}_2), \exists v' \in \mathcal{I}(\mathcal{Y}_1)\ s.t.\ v' = c(\ldots)\} \cup$
  $\{v \mid v \in \mathcal{I}(\mathcal{Y}_3), \exists v' \in \mathcal{I}(\mathcal{Y}_1)\ s.t.\ v' \neq c(\ldots)\}$
  provided:
  1. if $c(v_1, \ldots, v_n) \in \mathcal{I}(\mathcal{Y}_1)$ then $v_i \in \mathcal{I}(\mathcal{W}_i),\ 1 \le i \le n$
  2. if $v \in \mathcal{I}(\mathcal{Y}_1)$ and $v \neq c(\ldots)$ then $v \in \mathcal{I}(\mathcal{W})$

Figure 3. Set expressions

We define set expressions and their meanings in Figure 3. We have three kinds of set expressions. First, variable set expressions denote program points. Two special set variables are used to handle flows generated by function applications. $dom(\lambda x.e)$ denotes actual arguments flowing into $\lambda x.e$'s formal argument. $ran(\lambda x.e)$ denotes result values of function application. Second, abstractions and constructed value expressions denote set of values, which we call atomic set expressions. Lastly, *apply* and *case* expression denote semantics of the language. Meaning of set expression is given by extending interpretation $\mathcal{I}$, which is a mapping from set variables to sets of values. Note that we don't have environments as function closure because single predefined set environment is used. Restrictions in the extension guarantees that $\mathcal{I}$ is sound approximation of the program.

Set constraints are of the form $\mathcal{X} \supseteq se$ meaning that program point $\mathcal{X}$ contains the values denoted by se.

Set constraints are generated from the source program by the rules in Figure 4. Environment $\mathcal{E}$ maps program variables to corresponding set variables. The relation $E \vdash e \rhd (\mathcal{X}, C)$ says that in environment E, constraints C are

generated with new set variable $\mathcal{X}$ denoting program point e.

3.2. Set Constraint Solving

We present set constraint solving rules, **E**, in Figure 5. **E** simulates the data flows in programs by propagating set constraints. For example, for a function call site, rule **E1** links the result of the site with body of functions flowing into the site and rule **E2** links formal argument of those functions to actual argument of the site. Further simplifications are initiated by flowing values through links established.

We define analysis as the reflexive transitive closure $\mathbf{E}^*(C)$ of $\mathbf{E}(C)$, a single step of adding constraints by applying **E** to C. The closure is the set of initial constraints C and all added constraints by repeatedly applying **E**. The main result of set-based analysis[3] showed that we get explicit representation of safe approximation of the program by collecting all constraints of the form $\mathcal{X} \supseteq ae$ in $\mathbf{E}^*(C)$.

$$\mathcal{E} \vdash x \triangleright (\mathcal{E}(x),\ \{\})$$

$$\mathbf{E1}\quad \frac{\mathcal{E} \vdash e_1 \triangleright (\mathcal{X}_1,\ C_1) \quad \mathcal{E} \vdash e_2 \triangleright (\mathcal{X}_2,\ C_2) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq ran(\lambda x.e)}$$

$$\mathbf{E2}\quad \frac{\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

$$\mathbf{E3}\quad \frac{\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots)}{\mathcal{X} \supseteq \mathcal{Y}_2}$$

$$\mathbf{E4}\quad \frac{\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots, \mathcal{W}_i, \dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots, \mathcal{Z}_i, \dots)}{\mathcal{W}_i \supseteq \mathcal{Z}_i}$$

$$\mathbf{E5}\quad \frac{\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{X} \supseteq \mathcal{Y}_3}$$

$$\mathbf{E6}\quad \frac{\mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{W} \supseteq c'(\dots)}$$

$$\mathbf{E7}\quad \frac{\mathcal{X} \supseteq \mathcal{X}' \quad \mathcal{X}' \supseteq ae}{\mathcal{X} \supseteq ae}$$

Figure 5. Exhaustive solving (E)

4. Bi-directional Demand-Driven Set-Based Analysis

In this section, we develop a demand-driven set-based analysis by modifying Heintze's exhaustive set-based analysis presented in Section 3. Our demand-driven set-based analysis solves from arbitrary initial demands of interested program points, only those constraints affecting them and gives the same

results as exhaustive set-based analysis for them.

4.1. Making it Demand-Driven

Let's consider the rule **E1** of exhaustive solving **E** in Figure 5.

$$\frac{\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq ran(\lambda x.e)}$$

Here, we add the constraint that $\mathcal{X}$ contains $ran(\lambda x.e)$. We need this constraint if we want to know what values program point $\mathcal{X}$ evaluates to, i.e., if we have demand for $\mathcal{X}$. So, we need the following demand-driven rule with the notation $D(\mathcal{X})$ meaning that we have demand for $\mathcal{X}$.

$$\frac{D(\mathcal{X}) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq ran(\lambda x.e)}$$

Next step is adding new demands to ensure that we solve all necessary constraints. Let's continue considering the case **E1**. We have to ensure that all premises in the original rule **E1** are added in demand-driven solving. All constraints with $apply(\cdots)$ are directly generated from the source program. So, the premise, $\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)$ is guaranteed to be in demand-driven solving. However, the other premise, $\mathcal{X}_1 \supseteq \lambda x.e$, can be added during solving process. We can guarantee that this premise is added in demand-driven solving with $D(\mathcal{X}_1)$, i.e., we need to know what values $\mathcal{X}_1$ evaluates to. So, we can write the following demand-add rule.

$$\frac{D(\mathcal{X}) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)}{D(\mathcal{X}_1)}$$

Note that from $D(\mathcal{X}_1)$, we can identify $\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)$ right away because it is a generated constraint.

Demand-add rules specify solving sequence. For example, the rule we have just defined specifies solving sequence as follows: To solve constraints for $\mathcal{X}$ which is a call site of $\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)$, we first find all functions $\mathcal{X}_1$ evaluates to.

Using similar approach, we can make all rules demand-driven and get associated demand-add rules except for the case **E2** $dom(\lambda x.e)$, which we will discuss next.

4.2. Demand for Formal Argument

Let's consider the rule **E2**.

$$\frac{\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

We can make the rule itself demand driven as follows:

$$\frac{D(dom(\lambda x.e)) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

However, we can't get the associated demand-add rule as other cases. From $D(dom(\lambda x.e))$, we have only the clue that $\lambda x.e$ flows into some $\mathcal{X}_1$. Finding out all such $\mathcal{X}_1$ from $\lambda x.e$ is not trivial because $\mathcal{X}_1 \supseteq \lambda x.e$ might be a newly added constraint during solving.

Naive approach of following demand-add rule

works, but it is not fully demand-driven in the sense that we solve all call sites regardless of initial demands:

$$\frac{D(dom(\lambda x.e)) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)}{D(\mathcal{X}_1)}$$

In order to avoid this too loose demand-add rule, we need to consider solving constraints in the opposite direction. The information needed here is all call sites that $\lambda x.e$ flows into. So, we make it fully demand-driven by solving $\lambda x.e$ in the opposite backward direction. We specify solving directions with the notations $FD(\mathcal{X})$ for ordinary forward solving of $\mathcal{X}$ and $BD(se)$ for the opposite backward solving of se. With these two solvings, demand-add rule for forward formal argument is stated as follows:

$$\frac{FD(dom(\lambda x.e))}{BD(\lambda x.e)}$$

That is, we need to know where $\lambda x.e$ flows into (backward needness) if we need to know what values flow into formal argument x (forward needness).

4.3. Bi-directional Demand-Driven Solving

Our demand-driven solving rules consist of forward rules and backward rules interleaved. We solve some constraints in forward direction and some in backward direction. For the presentational simplicity, we include forward demand $FD(\mathcal{X})$ and backward demand $BD(ae)$ or $BD(\mathcal{X})$ as special kinds of set constraints.

Forward demand-driven solving, **F**, finds all constraints of the form $\mathcal{X} \supseteq se$ if we have the forward demand, $FD(\mathcal{X})$. The solving rules and demand-add rules of forward solving are shown in Figure 6. Two rules are indexed with the same number for solving rule and corresponding demand-add rule.

Backward demand-driven solving, **B**, finds all constraints of the form $\mathcal{X} \supseteq se$ if we have the backward demand, $BD(se)$. We present solving rules and demand-add rules in Figure 7. We formulated these rules similar to the way we did for the forward solving. For example, **B2** is derived from **E2** by adding the premise $BD(\mathcal{X}_2)$ to ensure that we add $dom(\lambda x.e) \supseteq \mathcal{X}_2$ only if we have the backward needness for $\mathcal{X}_2$ as follows:

$$\frac{BD(\mathcal{X}_2) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

To guarantee that non-trivial premise $\mathcal{X}_1 \supseteq \lambda x.e$ is included in the backward solving, we devise following demand-add rule $\mathbf{B2}^D$:

$$\frac{BD(\mathcal{X}_2) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)}{FD(\mathcal{X}_1)}$$

This rule specifies that to consider all program points that an actual argument of a call site flows into, we have to first find out all functions flowing into the call site. Other rules are derived according to similar ideas.

Demand-driven set-based analysis repeatedly

$$\textbf{F1}\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq ran(\lambda x.e)}$$

$$\textbf{F2}\quad \frac{FD(dom(\lambda x.e)) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

$$\textbf{F3}\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots)}{\mathcal{X} \supseteq \mathcal{Y}_2}$$

$$\textbf{F4}\quad \frac{FD(\mathcal{W}_i) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots, \mathcal{W}_i, \dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots, \mathcal{Z}_i, \dots)}{\mathcal{W}_i \supseteq \mathcal{Z}_i}$$

$$\textbf{F5}\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{X} \supseteq \mathcal{Y}_3}$$

$$\textbf{F6}\quad \frac{FD(\mathcal{W}) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{W} \supseteq c'(\dots)}$$

$$\textbf{F7}\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq \mathcal{X}' \quad \mathcal{X}' \supseteq ae}{\mathcal{X} \supseteq ae}$$

$$\textbf{F1}^D\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)}{FD(\mathcal{X}_1)}$$

$$\textbf{F2}^D\quad \frac{FD(dom(\lambda x.e))}{BD(\lambda x.e)}$$

$$\textbf{F3,5}^D\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)}{FD(\mathcal{Y}_1)}$$

$$\textbf{F4}^D\quad \frac{FD(\mathcal{W}_i) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots, \mathcal{W}_i, \dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)}{FD(\mathcal{Y}_1)}$$

$$\textbf{F6}^D\quad \frac{FD(\mathcal{W}) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)}{FD(\mathcal{Y}_1)}$$

$$\textbf{F7}^D\quad \frac{FD(\mathcal{X}) \quad \mathcal{X} \supseteq \mathcal{X}'}{FD(\mathcal{X}')}$$

Figure 6. Forward demand-driven solving (F)

applies rules **F** and **B**. Let C' be the union of the generated constraints and initial forward demands. We define $\mathbf{FB}(C')$ to be newly added constraints by applying **F** and **B** to C' once. Then, solving process is represented by the reflexive transitive closure, $\mathbf{FB}^*(C')$.

5. Equivalence

In this section, we prove that for the program points we want analyze, our demand-driven set-based analysis gives the

$$\textbf{B1}\quad \frac{BD(ran(\lambda x.e)) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq ran(\lambda x.e)}$$

$$\textbf{B2}\quad \frac{BD(\mathcal{X}_2) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$$

$$\textbf{B3}\quad \frac{BD(\mathcal{Y}_2) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots)}{\mathcal{X} \supseteq \mathcal{Y}_2}$$

$$\textbf{B4}\quad \frac{BD(\mathcal{Z}_i) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots, \mathcal{W}_i, \dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c(\dots, \mathcal{Z}_i, \dots)}{\mathcal{W}_i \supseteq \mathcal{Z}_i}$$

$$\textbf{B5}\quad \frac{BD(\mathcal{Y}_3) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{X} \supseteq \mathcal{Y}_3}$$

$$\textbf{B6}\quad \frac{BD(c'(\dots)) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3) \quad \mathcal{Y}_1 \supseteq c'(\dots) \quad c \neq c'}{\mathcal{W} \supseteq c'(\dots)}$$

$$\textbf{B7}\quad \frac{BD(ae) \quad \mathcal{X} \supseteq \mathcal{X}' \quad \mathcal{X}' \supseteq ae}{\mathcal{X} \supseteq ae}$$

$$\textbf{B1}^D\quad \frac{BD(ran(\lambda x.e))}{BD(\lambda x.e)}$$

$$\textbf{B2}^D\quad \frac{BD(\mathcal{X}_2) \quad \mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2)}{FD(\mathcal{X}_1)}$$

$$\textbf{B3}^D\quad \frac{BD(\mathcal{Y}_2) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)}{FD(\mathcal{Y}_1)}$$

$$\textbf{B4}^D\quad \frac{BD(\mathcal{Z}_i)}{BD(c(\dots, \mathcal{Z}_i, \dots))}$$

$$\textbf{B5}^D\quad \frac{BD(\mathcal{Y}_3) \quad \mathcal{X} \supseteq case(\mathcal{Y}_1, c(\dots) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)}{FD(\mathcal{Y}_1)}$$

$$\textbf{B7}^D\quad \frac{BD(ae) \quad \mathcal{X}' \supseteq ae}{BD(\mathcal{X}')}$$

Figure 7. Backward demand-driven solving (B)

same results as the original exhaustive analysis. We prove the equivalence by using similar strategy and notations Heintze used to prove equivalence of his demand-driven pointer analysis [6].

Equivalence guaranteed by our demand-driven formulation is restricted to top-level structure in case of constructed values. Suppose we have forward demand for $\mathcal{X}$ and constraint $\mathcal{X} \supseteq c(\mathcal{X}_1)$ is in the solution. In this case, values for $\mathcal{X}_1$ are not guaranteed to be in the solution. This is not surprising since we only guarantees that all constraints of the form $\mathcal{X} \supseteq se$ are in the solution and we don't have

constructed value set expression of the form $c(ae)$. Solution of set-based analysis is a regular tree grammar specifying how to build values in the solution [4].

This is not severe restriction. First, one may actually need to know only top-level structure of values. In those situations, our method is more efficient than finding values for all components together. Second, if values for components are needed, those values can be solved by invoking the analysis again with demands for needed components. In reinvocations, constraints solved in previous invocations are saved and need not be solved again. So there's not much performance loss by solving constraints through successive steps for components.

Now, we prove the equivalence. Let's first define following assertions to represent result constraints of demand driven solving. Let C be generated constraints and C' be the union of generated constraints and initial forward demands.

- $\mathcal{X} \vdash_F \mathcal{X} \supseteq se$
  this assertion states if $FD(X)$ in $\mathbf{FB}^*(C')$ then $X \supseteq se \in \mathbf{FB}^*(C')$.

- $se \vdash_B \mathcal{X} \supseteq se$
  this assertion states if $FD(X)$ in $\mathbf{FB}^*(C')$ then $X \supseteq se \in \mathbf{FB}^*(C')$.

Lemma 1 (Soundness). *If* $\mathcal{X} \supseteq se \in \mathbf{FB}^*(C')$ *then* $\mathcal{X} \supseteq se \in \mathbf{E}^*(C)$.

Proof: This is trivial since demand only restricts solving process. We start demand-driven solving with the same value constraints, and all solving rules in **F** and **B** are restricted version of the same rules in **E** each with an additional premise for demand. □

Lemma 2 (Completeness). *If* $\mathcal{X} \supseteq se \in \mathbf{E}^*(C)$ *then* $\mathcal{X} \vdash_F \mathcal{X} \supseteq se$ and $se \vdash_B \mathcal{X} \supseteq se$

Proof: We prove by induction on the proof tree of derivations of $\mathcal{X} \supseteq se \in \mathbf{E}^*(C)$.

– Base cases

Base cases of $\mathcal{X} \supseteq se \in \mathbf{E}^*(C)$ are generated constraints $\mathcal{X}$. All these constraints are included in forward (backward) solving since $C' \supseteq C$.

– $\dfrac{\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{dom(\lambda x.e) \supseteq \mathcal{X}_2}$

$\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \in \mathbf{FB}^*(\mathcal{C}')$ (base case)

i) $dom(\lambda x.e) \vdash_F dom(\lambda x.e) \supseteq \mathcal{X}_2$
We assume $FD(dom(\lambda x.e)) \in \mathbf{FB}^*(\mathcal{C}')$.
$\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \in \mathbf{FB}^*(\mathcal{C}')$
(base case)
$BD(\lambda x.e) \in \mathbf{FB}^*(\mathcal{C}')$
(by $\mathbf{F2}^D$)
$\mathcal{X}_1 \supseteq \lambda x.e \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $\lambda x.e \vdash_B \mathcal{X}_1 \supseteq \lambda x.e$)
$dom(\lambda x.e) \supseteq \mathcal{X}_2 \in \mathbf{FB}^*(\mathcal{C}')$
(by **F2** applied to above results)

ii) $\mathcal{X}_2 \vdash_B dom(\lambda x.e) \supseteq \mathcal{X}_2$
We assume $BD(\mathcal{X}_2) \in \mathbf{FB}^*(\mathcal{C}')$.
$\mathcal{X} \supseteq apply(\mathcal{X}_1, \mathcal{X}_2) \in \mathbf{FB}^*(\mathcal{C}')$
(base case)
$FD(\mathcal{X}_1) \in \mathbf{FB}^*(\mathcal{C}')$
(by $\mathbf{B2}^D$)
$\mathcal{X}_1 \supseteq \lambda x.e \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $\mathcal{X}_1 \vdash_F \mathcal{X}_1 \supseteq \lambda x.e$)
$dom(\lambda x.e) \supseteq \mathcal{X}_2 \in \mathbf{FB}^*(\mathcal{C}')$
(by **B2** applied to above results)

– $\dfrac{\mathcal{X} \supseteq \mathcal{X}' \quad \mathcal{X}' \supseteq ae}{\mathcal{X} \supseteq ae}$

i) $\mathcal{X} \vdash_F \mathcal{X} \supseteq ae$
We assume $FD(\mathcal{X}) \in \mathbf{FB}^*(\mathcal{C}')$.
$\mathcal{X} \supseteq \mathcal{X}' \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $\mathcal{X} \vdash_F \mathcal{X} \supseteq \mathcal{X}'$)
$FD(\mathcal{X}') \in \mathbf{FB}^*(\mathcal{C}')$
(by $\mathbf{F7}^D$ applied to above results)
$\mathcal{X}' \supseteq ae \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $\mathcal{X}' \vdash_F \mathcal{X}' \supseteq ae$)
$\mathcal{X} \supseteq ae \in \mathbf{FB}^*(\mathcal{C}')$
(by **F7** applied to above results)

ii) $ae \vdash_B \mathcal{X} \supseteq ae$
We assume $BD(ae) \in \mathbf{FB}^*(\mathcal{C}')$.
$\mathcal{X}' \supseteq ae \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $ae \vdash_B \mathcal{X}' \supseteq ae$)
$BD(\mathcal{X}') \in \mathbf{FB}^*(\mathcal{C}')$
(by $\mathbf{B7}^D$ applied to above results)
$\mathcal{X} \supseteq \mathcal{X}' \in \mathbf{FB}^*(\mathcal{C}')$
(by I.H. $\mathcal{X}' \vdash_B \mathcal{X} \supseteq \mathcal{X}'$)
$\mathcal{X} \supseteq ae \in \mathbf{FB}^*(\mathcal{C}')$
(by **B7** applied to above results)

– Other cases are proved similarly. □

Theorem 1 (Equivalence). *For all* $FD(\mathcal{X}) \in \mathcal{C}$, $\mathcal{X} \supseteq se \in \mathbf{E}^*(C)$ *iff* $\mathcal{X} \supseteq se \in \mathbf{FB}^*(C)$.

Proof: It follows immediately from Lemma 1 and Lemma 2

6. Experimental Results and Discussion

We have implemented prototypes of both exhaustive analysis and demand-driven analysis over nML programming language system [7] to measure the performance of demand-driven analysis. We analyzed source code after pattern matching compilation based on decision tree model [1], which transforms ML's complex pattern matching to the restricted form of the core language. For other parts of full ML not represented in core language, we extended the core analysis with similar ideas in [5] except that we didn't analyze arithmetics.

We have experimented by solving each program point separately. For each program point, we start analysis from generated constraints from source program and with single forward demand for the point. As mentioned in Section 5, for the initial program point of interest, we solved for full structure of values by reinvocating the analysis for the components of structured values. We included significant libraries such as list libraries in the source program and analyzed them together.

Summary of experimental results are shown in Figure 8. First four programs are widely used benchmarks for ML. `dsba` is our analyzer itself. `patcomp` is a pattern match compiler for nML. `evalcps` is an interpreter and continuation passing style (cps) converter of simple functional language. Initial constraints are generated constraints from the program roughly indicating program size. We show maximum and average percentages of

Program (lines)	Initial[a] constraints	Ex[b] solving	DD solving(%) [c] Max. [d]	Avg. [e]
`fft` (238)	1348	1982	44.0	5.2
`nucleic` (3398)	15770	18574	36.9	0.8
`kb` (500)	3826	140518	83.9	33.7
`lexgen` (1282)	9035	50285	72.3	23.1
`dsba` (3162)	13740	115087	77.9	31.4
`patcomp` (1406)	6609	350493	84.0	39.5
`evalcps` (604)	2987	21136	85.8	21.2

[a] Initial generated constraints from source program
[b] Constraints added during exhaustive solving
[c] Demand-driven solving of each program points separately
[d] Maximum percentage of constraints added compared to exhaustive solving
[e] Average percentage of constraints added compared to exhaustive solving

Figure 8. Experimental results (summary)

	`fft`	`nucleic`	`kb`	`lexgen`	`dsba`	`patcomp`	`evalcps`
0-5 [a]	86.3 [b]	94.2	56.0	64.3	56.5	50.5	67.3
5-10	0.2	1.8	0	0	0	0	1.9
10-15	1,9	1.8	0	0	0	0	0
15-20	0.1	2.1	0	0	0	0	0
20-25	0	0	0	0	0	0	0
25-30	0	0	0	0	0	0	0
30-35	3.9	< 0.1 [c]	0	0	0	0	4.3
35-40	0	0.1	0	0	0	0	0.1
40-45	7.3	0	0	0	0	0	1.8
45-50	0	0	0	0	0	0	2.8
50-55	0	0	0	0	0	0	3.4
55-60	0	0	0	0	0	0	0.1
60-65	0	0	0	12.6	0	0	0
65-70	0	0	0	23.0	20.6	0	0
70-75	0	0	0	0.1	3.6	9.1	0
75-80	0	0	40.4	0	19.4	0	0
80-85	0	0	3.6	0	0	40.4	17.6
85-90	0	0	0	0	0	0	0.6
90-95	0	0	0	0	0	0	0
95-100	0	0	0	0	0	0	0

[a] Range of percentage of constraints added compared to exhaustive solving
[b] Percentage of program points belonging to the range
[c] Less than 0.1%

Figure 9. Experimental results (details)

constraints added for demand-driven solving of a program point compared to exhaustive solving. Maximum percentage indicates the program point solving the largest number of constraints. We show average of all program points though it may not be much meaningful because of high variance.

More detailed results are shown in Figure 9. For each program, we give percentages of program points solving 0-5%, ⋯, 95-100% of constraints compared to exhaustive solving. We could see some interesting points with the results.

– Many program points solve 0-5%. This

behavior is quite natural since many program points do simple computation. For example, in extreme, program points for constants solve 0\%.

– Many program points solve near maximum percentages. In mono-variant analysis, all function calls are merged. So, if the result of a function call is needed, we have to analyze all other call sites of the function, which can be the cause of this behaviour. Also, we can think that mono-variance is causing clustering of program points with many 0's in Figure 9.

– 44%, 36.9% maximum percentages for `fft` and `nucleic`. What we observed from this behaviour is that demand-driven formulation reflects crude approximation done in base analysis and doesn't solve unnecessary constraints due to those approximation. `fft` and `nucleic` are intensive numerical applications. However, we didn't analyzed arithmetics. Analysis result for $e_1 + e_2$ is $\top$, meaning all values are possible. So, values for e_1 and e_2 are not needed to get this $\top$ value. When we modified demand-driven formulation to enforce demand for e_1 and e_2, we got 61.4% maximum percentage for `fft` and 89.8% for `nucleic`. Also, `fft` uses arrays intensively. We analyzed arrays by collapsing all cells of an array into a single reference cell as Heintze did in [5]. Again, this approximation discards any need for array indexes other than needs for themselves. When we also enforced demands for array indexes in addition to arithmetic arguments, we got 80% maximum percentage.

7. Conclusion

We have developed a demand-driven approach for set-based analysis. By incorporating demands to the analysis, only those constraints related to interested program points are solved. To get fully demand-driven formulation, we solved some constraints in forward direction and some in backward direction. We have proved that our approach is equivalent to original exhaustive approach in the sense that for the initial demands, our approach gives exactly the same results. We have implemented prototypes of our analysis and experimental results show interesting aspects of demand-driven analysis.

References

[1] Marianne Baudinet and David MacQueen, "Tree pattern matching for ML", unpublished paper, 1985.

[2] Sandip K. Biswas, "A demand-driven set-based analysis", *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 372-385, 1997.

[3] Nevin Heintze, "Set-Based Program Analysis", PhD thesis, Carnegie Mellon University, October 1992.

[4] Nevin Heintze, "Set constraints in program analysis, *Proceedings of International Symposium on Logic Programming*, 1993.

[5] Nevin Heintze, "Set-based analysis of ML programs, *Proceedings of the ACM*

Conference on Lisp and Functional Programming, pages 306-317, 1994.

[6] Nevin Heintze and Olivier Tardieu, "Demand driven pointer analysis", *Proceedings of ACM Conference on Programming Language Design and Implementation*, pages 24-34, 2001.

[7] nML programming language system, version 0.92a, Research On Program Analysis System, KAIST, March 2002, http://ropas.kaist.ac.kr/n.

Woongsik Choi

2002, M.S. Computer Science, KAIST

2000, B.S. Computer Science, KAIST

Kwangkeun Yi

1995 - now Associate Professor, KAIST

1993 - 1995 Member of Technical Staff, Software Principles Research Dept., Bell Labs., Murray Hill

1993, Ph.D. Computer Science, Univ. of Illinois at Urbana-Champaign

1990, M.S. Computer Science, Univ. of Illinois at Urbana-Champaign

1987, B.S. Computer Science & Statistics, Seoul National University