ROPAS Research On Program Analysis System
National Creative Research Initiative Center
Korea Advanced Institute of Science and Technology

ROPAS MEMO
1999-1
November 17, 1999

Preparing Set-Based Analysis for Run-time Specialization[1]

Hyunjun Eo and Kwangkeun Yi
{poisson,kwang}@ropas.kaist.ac.kr

Abstract

We present a technique of using static analysis for estimating program's input-dependent properties. A static analysis that is originally designed for estimating the input-*independent* properties of programs is transformed into one that can safely estimate the input-*dependent* properties at the programs' input occurrence. No profile is collected and no probing codes inside the running program are needed.

Our idea is to defer the finish of the static analysis to the program's run-time. By analyzing the static analysis, we identify the parts of the analysis that are sensitive to the program's inputs, hence need to be deferred to the program's run-time. Then by using an analysis named *static value-slicing*, we short-cut some of the dynamic parts so that they are solved by simple membership tests for the program's input. This re-formulation accelerates the analysis; once the program's input occurs the prepared dynamic parts can immediately and simultaneously start to resolve.

Every step of our technique is formally defined and proven correct.

1 Introduction

1.1 Motivation

Program's properties that are invariant to its inputs are necessary but not sufficient for optimizations. *Input-dependent* program properties are also needed if we want to tailor a program for its popular inputs.

A conventional approach to achieving the input-dependent properties of a program is profiling. Profile-based optimizations use this statistics to tailor the target code to the popular inputs.

But the profile-based analysis has some difficulties. The properties from the profile data are not elaborate enough for some optimizations. Usually the collected statistics are frequency countings of visiting particular control points (or paths) in the program. This hot-spot information is useful for optimizations that are sensitive, for example, to correct branch predictions. But for other optimizations like parallelizations, we need the read/write behaviors of the program variables, whose monitoring needs more than frequency countings. Furthermore, collecting such elaborate run-time properties will have too much overhead on the running programs, given that the current hot-path profiling has an overhead of 15%-30% [BL94, BMS98]. As the probing codes embedded inside the program to profile become more complicated, it becomes harder to strike the balance between the profiling overhead and its precision.

We need more general techniques for estimating, without profiling, input-dependent properties of programs. The situation becomes more demanding in the global and web computing environment, where a large number of code consumers (users) are apart from the code producer (compiler) over the network. The code producer compiles the source program and transmits

[1]This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

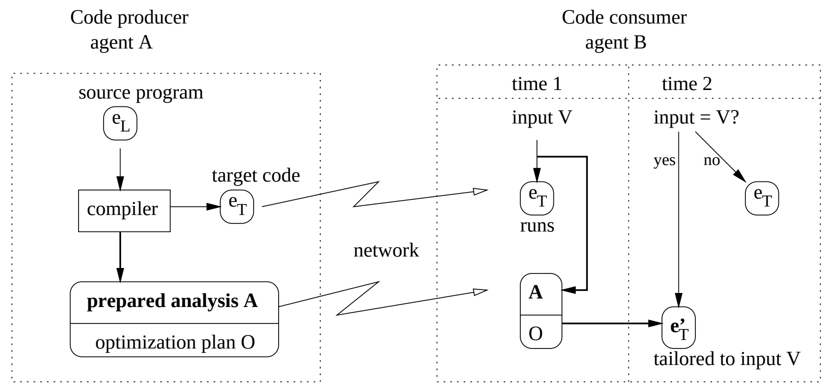

Figure 1: The scenario

the compiled code to the consumers at the other ends of the network for execution. The job of input-dependent optimizations (tailoring the transmitted code to some popular inputs) is delegated to each consumer. (Without this delegation, the code producer will be swamped with tailoring codes for each of the large number of consumers.) Because the consumers have no access to the source program, the optimizations on the transmitted compiled code are limited. The code producer, on the other hand, is eager to supply aggressive optimizations that can adjust to each customer's typical inputs. Such aggressive optimizations need elaborate program analyses usually at the source-level.

Therefore, the following scenario is anticipated. The code producer prepares an analysis that can estimate, at the consumer site, the input-dependent properties of the source. The code producer also has the knowledge on how an optimized target code can be generated from the estimated run-time properties of the source. The code producer transmits to the consumer these two things, a "prepared analysis A for the source" and an "optimization plan O for the target" (see Figure 1), together with the compiled target code. The code consumer executes the transmitted code. When an input occurs the prepared analysis A estimates the dynamic properties of the source, independent of the running code. The analysis result will activate the optimization plan O to generate an aggressively adjusted target code. When the same or similar input occurs, the consumer dispatches the adjusted code instead of the original one.

1.2 Overview

This article presents a technique for designing such a "prepared analysis A," an analysis that estimates the input-dependent properties of the programs. The technique is based on the static analysis framework. A static analysis that is originally designed for estimating the input-independent properties of programs is transformed into one that can safely estimate the input-dependent properties at the programs' input occurrence. No profile (or trace) is collected and no probing codes inside the running program are needed.

Our idea of using a static analysis for estimating the dynamic properties of programs is simply to defer the finish of the static analysis to the program's run-time. By analyzing the static analysis, we identify the parts of the analysis that are sensitive to the program's inputs, hence need to be deferred to the program's run-time. Then by using a technique named *static value-slicing*, we short-cut some of the dynamic parts so that they are solved by simple membership tests on the program's input. This re-formulation prepares the dynamic parts for their fast resolution; once the program's input occurs the prepared dynamic parts can immediately and simultaneously start to resolve.

For example, consider an analysis for estimating the values of the following expression, which depends on an input variable α:

$$\texttt{if } e \texttt{ = 0 then 1 else 2}.$$

The expression's values $\mathcal{X}$ as conditional set constraints are:

$$\begin{array}{rcl} \mathcal{X}_e \supseteq \{0\} & \Rightarrow & \mathcal{X} \supseteq \{1\} \\ \mathcal{X}_e \supseteq \{n|n \neq 0\} & \Rightarrow & \mathcal{X} \supseteq \{2\}. \end{array}$$

Suppose the condition expression e is input-dependent. Then the compiler has to conclude that the `if` expression's value is $\{1,2\}$ because the condition expression's value is unknown at compile-time. At run-time, depending on the value of $\mathcal{X}_e$, the conditional analysis can determine a sharper result. For making it better, we can re-formulate the conditions $\mathcal{X}_e \supseteq \{0\}$ and $\mathcal{X}_e \supseteq \{n|n \neq 0\}$ as the input α's membership tests for some sets V and W, respectively:

$$\begin{array}{rcl} \alpha \in V & \Rightarrow & \mathcal{X} \supseteq \{1\} \\ \alpha \in W & \Rightarrow & \mathcal{X} \supseteq \{2\} \\ \text{else} & \Rightarrow & \mathcal{X} \supseteq \{1,2\}. \end{array}$$

At run-time when an input ν to the program occurs, the conditions immediately resolve into true or false without the delay for computing the $\mathcal{X}_e$. Thus the expression's value set $\mathcal{X}$ specific to input ν can be quickly estimated. Note that because the two exclusive sets V and W are estimated at compile-time (by yet another static analysis) they are not necessarily exhaustive. Some inputs can be outside of $V \cup W$, in which case the estimated dynamic properties becomes $\{1,2\}$, as inaccurate as static analysis.

1.3 Organization

Section 2 presents the source language L. Section 3 presents our static analysis for L programs that have input variables. From this analysis, we define a dynamic analysis that estimates the input-dependent properties of the programs. Section 4 and 5 present techniques of moving the overhead of the dynamic analysis to the compile-time of the programs. Section 4 presents a technique of identifying and deferring the input-dependent parts of the analysis, while solving its input-independent parts as far as possible. Section 5 presents a technique of short-cutting some of the dynamic parts, in order to further accelerate the remaining dynamic analysis parts. Section 6 presents an example. Section 7 discusses related works. Section 8 concludes.

2 The Source Language L

We consider a call-by-value, higher-order language, which can be considered a core of ML[MTHM97]. Expressions in the language are either variables, functions, function applications, data constructions, data de-constructions, or conditional branches: see Figure 2. We assign identifying labels to every sub-expression. Values are either functions or data. A datum value $\kappa(v)$ is constructed by "`con` κ e" from a data constructor κ and an expression e for the constructor's argument v. The argument value v is recovered by "`decon` κ e" where e computes $\kappa(v)$. Switch expression "`case` e_1 κ e_2 e_3" branches to e_2 or e_3 depending on e_1's data constructor.

We define the semantics (Figure 2) using the evaluation contexts technique [Fea87, FF97]. The evaluation context defines a left-to-right, call-by-value reduction. We write $\mathcal{E}[e]$ if the

Syntax:

$$
\begin{array}{rrcll}
\textit{Expr} & e & ::= & e_l \mid x \mid v \mid e\ e \mid \texttt{con}\ \kappa\ e & \text{labeled expressions} \\
& & \mid & \texttt{decon}\ \kappa\ e \mid \texttt{case}\ e\ \kappa\ e\ e & \\
\textit{Val} & v & ::= & \lambda x.e \mid \kappa(v) & \text{values} \\
\textit{Con} & \kappa & & & \text{data constructors} \\
\textit{Var} & x, f & & & \text{variables} \\
\textit{Label} & l & & & \text{expression labels}
\end{array}
$$

Semantics:

$$\textit{eval}(e) = e \longmapsto e' \longmapsto \cdots.$$

Reduction Rules:

$$
\begin{array}{rcll}
\mathcal{E}[(\lambda x.e)\ v] & \longmapsto & \mathcal{E}[\ [v/x]e\ ] & \\
\mathcal{E}[\texttt{con}\ \kappa\ v] & \longmapsto & \mathcal{E}[\kappa(v)] & \\
\mathcal{E}[\texttt{decon}\ \kappa\ \kappa(v)] & \longmapsto & \mathcal{E}[v] & \\
\mathcal{E}[\texttt{case}\ \kappa(v)\ \kappa\ e_1\ e_2] & \longmapsto & \mathcal{E}[e_1] & \\
\mathcal{E}[\texttt{case}\ \kappa'(v)\ \kappa\ e_1\ e_2] & \longmapsto & \mathcal{E}[e_2] & \\
\mathcal{E}[v_l] & \longmapsto & \mathcal{E}[v] & \text{unlabel}
\end{array}
$$

Evaluation Contexts:

$$\mathcal{E}\ ::=\ [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{con}\ \kappa\ \mathcal{E} \mid \texttt{decon}\ \kappa\ \mathcal{E} \mid \texttt{case}\ \mathcal{E}\ \kappa\ e\ e \mid \mathcal{E}_l$$

Figure 2: The source language L: syntax and semantics

hole in context $\mathcal{E}$ is filled with e. Note that the hole can be surrounded by labels: $\mathcal{E}_l$. The reduction rules are conventional, except that they preserve the labels of expressions during reductions and the *unlabel* rule removes the label from an expression when its value is needed. As usual, the substitution operation $[v/x]e$ replaces every free occurrence of x in e by v. Here the substitution also preserves the labels of x in e: for example, $[v/x]x_l = v_l$.

3 Start Point: A Set-Based Analysis

In this section, we will define a static analysis from which we will derive, in the subsequent sections, an analysis that estimates input-dependent properties of programs. The analysis estimates the values of expressions. We present it in the set-based analysis framework [Hei92, Hei93, HM97, AH95].

Set-based static analysis consists of two phases: collecting set constraints and solving them. The first phase derives constraints that describe the data flows between the expressions of the analyzed program. The second phase finds the sets of values that satisfy the constraints. A solution is a table from set variables in the constraints to the finite descriptions of such sets of values.

Each sub-expression e_l and program variable x has set variables $\mathcal{X}_l$ and $\mathcal{X}_x$, respectively representing expression's values and variable's bound values. Each set constraint is of the form $\mathcal{X} \supseteq se$, where $\mathcal{X}$ is a set variable and se is a set expression. The constraint indicates that the set $\mathcal{X}$ must have the set se. The set expression se has seven kinds, each of which corresponds to each program construct (see Figure 3). Semantics of set expressions naturally follows from their corresponding language constructs. For example, $\kappa\,\mathcal{X}_1$ represents a set of $\kappa(v)$'s where v is an element of $\mathcal{X}_1$. $app(\mathcal{X}_1, \mathcal{X}_2)$ represents the set of values returned from applications of functions in $\mathcal{X}_1$ to parameters in $\mathcal{X}_2$. $case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)$ indicates the values of $\mathcal{X}_2$ if a value in $\mathcal{X}_1$ is $\kappa(v)$ or the values of $\mathcal{X}_3$ otherwise. The formal semantics of set expressions is defined by

Syntax of set expressions:

$$
\begin{array}{rcll}
se & ::= & \mathcal{X}_i & \text{set variables} \\
 & | & \lambda x.e & \text{lambdas} \\
 & | & \kappa\ \mathcal{X}_i & \text{data sets} \\
 & | & \kappa^{-1}\ \mathcal{X}_i & \text{data argument sets} \\
 & | & app(\mathcal{X}_i, \mathcal{X}_i) & \text{sets from function call} \\
 & | & case(\mathcal{X}_i, \kappa, \mathcal{X}_i, \mathcal{X}_i) & \text{sets from switch} \\
 & | & \top & \text{universe set} \\
i & \in & \{l, x \mid e_l \in \mathit{Expr}, x \in \mathit{Var}\} & \text{indices for the sub-expressions} \\
 & & & \text{and variables}
\end{array}
$$

Semantics of set expressions:

$$
\begin{array}{rcl}
\mathcal{I}(\mathcal{X}_i) & \subseteq & Val \\
\mathcal{I}(\top) & = & Val \\
\mathcal{I}(\lambda x.e) & = & \{\lambda x.e\} \\
\mathcal{I}(\kappa\ \mathcal{X}_1) & = & \{\kappa(v) \mid v \in \mathcal{I}(\mathcal{X}_1)\} \\
\mathcal{I}(\kappa^{-1}\ \mathcal{X}_1) & = & \{v \mid \kappa(v) \in \mathcal{I}(\mathcal{X}_1)\} \\
\mathcal{I}(app(\mathcal{X}_1, \mathcal{X}_2)) & = & \{v \mid \lambda x.e \in \mathcal{I}(\mathcal{X}_1), v \in \mathcal{I}(\mathcal{X}_e), \mathcal{I}(\mathcal{X}_x) \supseteq \mathcal{I}(\mathcal{X}_2)\} \\
\mathcal{I}(case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)) & = & \{v \mid v \in \mathcal{I}(\mathcal{X}_2), \kappa(v') \in \mathcal{I}(\mathcal{X}_1)\} \cup \\
 & & \{v \mid v \in \mathcal{I}(\mathcal{X}_3), \kappa'(v') \in \mathcal{I}(\mathcal{X}_1), \kappa' \neq \kappa\}
\end{array}
$$

Rules $\wp \rhd \mathcal{C}$ for deriving constraints $\mathcal{C}$ from program $\wp$:

$$x_l \rhd \{\ \mathcal{X}_l \supseteq \mathcal{X}_x\}$$

$$\frac{e_1 \rhd \mathcal{C}_1}{(\texttt{con}\ \kappa\ e_1)_l \rhd \{\mathcal{X}_l \supseteq \kappa(\mathcal{X}_1)\} \cup \mathcal{C}_1} \qquad \frac{e_1 \rhd \mathcal{C}_1}{(\texttt{decon}\ \kappa\ e_1)_l \rhd \{\mathcal{X}_l \supseteq \kappa^{-1}(\mathcal{X}_1)\} \cup \mathcal{C}_1}$$

$$\frac{e_1 \rhd \mathcal{C}_1}{(\lambda x.e_1)_l \rhd \{\mathcal{X}_l \supseteq \lambda x.e_1\} \cup \mathcal{C}_1} \qquad \frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2}{(e_1\ e_2)_l \rhd \{\mathcal{X}_l \supseteq app(\mathcal{X}_1, \mathcal{X}_2)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{e_1 \rhd \mathcal{C}_1 \quad e_2 \rhd \mathcal{C}_2 \quad e_3 \rhd \mathcal{C}_3}{(\texttt{case}\ e_1\ \kappa\ e_2\ e_3)_l \rhd \{\mathcal{X}_l \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

Figure 3: Set constraints for set-based analysis: syntax, semantics, and specification rules

an interpretation $\mathcal{I}$ that maps from set expressions to sets of values (see Figure 3). We call an interpretation $\mathcal{I}$ a *model* (a solution) of a conjunction $\mathcal{C}$ of constraints if, for each constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$, $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$.

Our static analysis is defined to be the least model of set constraints. The constraint system guarantees the existence of the least model because every operator is monotonic (in terms of set-inclusion) and each constraint's left-hand-side is a single variable [Hei92].

3.1 Collecting the Initial Constraints

Figure 3 has the syntax-directed, linear rules for collecting the initial constraints $\mathcal{C}$ from a program $\wp$:

$$\wp \triangleright \mathcal{C}.$$

Each expression collects constraints from its sub-expressions and adds one or two constraints for itself that describe the data flows from the sub-expressions. For example, function application $e_1\ e_2$ collects two sets of constraints for its two sub-expressions and adds one constraint $\mathcal{X}_l \supseteq app(\mathcal{X}_1, \mathcal{X}_2)$ to describe that $\mathcal{X}_l$ has values returned from the functions in $\mathcal{X}_1$ (set variable for e_1) with its arguments in $\mathcal{X}_2$ (set variable for e_2).

3.2 Solving the Constraints

The solving phase closes the initial constraint set $\mathcal{C}$ under the rules S in Figure 4. Intuitively, the rules propagate values along all the possible data flow paths in the program. Each propagation rule dissolves compound set constraints into smaller ones, which approximates the steps of the value flows between expressions. Consider the rule for application result $\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2)$:

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e_3}{\mathcal{X} \supseteq \mathcal{X}_3 \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

It introduces $\mathcal{X} \supseteq \mathcal{X}_3$ if a function to call has body expression e_3, and if so, adds $\mathcal{X}_x \supseteq \mathcal{X}_2$ to simulate the parameter binding. (We will call such constraint ($\mathcal{X}_1 \supseteq \lambda x.e_3$) that triggers the dissolution of a compound set-constraints "firing constraint.") Other rules are similarly straightforward from the semantics of corresponding set expressions.

We write $A \vdash_R c$ if c is derivable from A using rules R, and write $R^*(A)$ for the closure of A under rules R, i.e., the set $\{c \mid A \vdash_R c\}$.[2] For a collection $\mathcal{C}$ of set constraints, we write $Vars(\mathcal{C})$ for the set of set variables in $\mathcal{C}$, and $\mathcal{C}(\mathcal{X})$ for the set $\{\mathcal{X} \supseteq se \mid \mathcal{X} \supseteq se \in \mathcal{C}\}$ of constraints for $\mathcal{X}$ in $\mathcal{C}$.

Among the set of constraints in $S^*(\mathcal{C})$, completely dissolved constraints ($atom(S^*(\mathcal{C}))$) constitute the least model of $\mathcal{C}$. We call such constraints "atomic." An atomic constraint is $\mathcal{X} \supseteq ae$ whose right-hand-side ae (atomic expression) explicitly denotes a value set:

$$ae \ ::= \ \lambda x.e \ \mid \ \kappa\, \mathcal{X} \ \mid \ \top.$$

Note that a set $G = \{\mathcal{X}_1 \supseteq ae_1, \cdots, \mathcal{X}_n \supseteq ae_n\}$ of atomic constraints denotes the set of sentences generated by its grammatical interpretation (a *regular tree grammar* where set variables are non-terminals) $\mathcal{X}_1 \ ::= \ ae_1, \cdots, \mathcal{X}_n \ ::= \ ae_n$. We write $[\![G]\!]$ for such set of sentences.

Theorem 1 *Let $\wp$ be a closed term and $\wp \triangleright \mathcal{C}$. The least model of $\mathcal{C}$ is* $\{\mathcal{X} \mapsto [\![atom(S^*(\mathcal{C})(\mathcal{X}))]\!] \mid \mathcal{X} \in Vars(\mathcal{C})\}$.

Proof. [Hei93, Hei92] □

[2] This closure is also defined as the least fixpoint $lfp\ (\lambda X.A \cup \{c \mid X \vdash_R c\})$.

$$\frac{\mathcal{X} \supseteq \kappa^{-1}\mathcal{Y} \quad \mathcal{Y} \supseteq \kappa\mathcal{Z}}{\mathcal{X} \supseteq \mathcal{Z}} \qquad \frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa\mathcal{Y}}{\mathcal{X} \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa'\mathcal{Y} \quad \kappa' \neq \kappa}{\mathcal{X} \supseteq \mathcal{X}_3}$$

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e_3}{\mathcal{X} \supseteq \mathcal{X}_3 \quad \mathcal{X}_x \supseteq \mathcal{X}_2} \qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

Figure 4: Rules S for solving set constraints

Definition 1 **(**sba**)** *Let* $\wp$ *be a closed term and let* $\wp \rhd \mathcal{C}$. $sba(\wp)(l) \stackrel{def}{=} [\![atom(S^*(\mathcal{C})(\mathcal{X}_l))]\!]$.

The set-based analysis $sba(\wp)$ is a safe approximation of the program values:

Theorem 2 (Safety of sba**)** *Let* $\wp$ *be a closed term and* $\wp \rhd \mathcal{C}$. *If* $\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ *then* $v \in sba(\wp)(l)$.

Proof. Following the proof strategies in [Hei93, Hei92], we use the set-based operational semantics [Hei93] as an intermediate step and Theorem 1. □

3.3 Analysis of Open Terms

So far we assume that the analyzed program has no input (free) variable. From now on, we assume that every program $\wp$ has one input variable α.

3.3.1 Static Analysis: Estimating Invariants

For program $\wp$ with an input variable α, the set $\mathcal{C}$ of the initial constraints ($\wp \rhd \mathcal{C}$) has no set constraint for $\mathcal{X}_\alpha$. Thus, set constraints that uses $\mathcal{X}_\alpha$ cannot be solved.

It is, however, straightforward to use $\mathcal{C}$ to achieve a static analysis of estimating the input-invariant properties. We start the solving process $S^*(\bullet)$ with the extra constraint $\mathcal{X}_\alpha \supseteq \top$ for the input variable α. Note that $\top$ indicates the universe set *Val* of values. The analysis process is:

let	$\wp \rhd \mathcal{C}$	set-up constraints at compile-time
in	$S^*(\mathcal{C} \cup \{\mathcal{X}_\alpha \supseteq \top\})$	solve constraints at compile-time

Because no information (universe) is assumed about the inputs, above analysis approximates properties that are invariant to the inputs.

Definition 2 **(**$sba_\top$**)** *Let* $\wp$ *be a term with a free variable* α *and let* $\wp \rhd \mathcal{C}$. $sba_\top(\wp)(l) \stackrel{def}{=} [\![atom(S^*(\mathcal{C} \cup \{\mathcal{X}_\alpha \supseteq \top\})(\mathcal{X}_l))]\!]$.

Theorem 3 (Safety of $sba_\top$**)** *For all* $\nu \in Val$, *if* $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ *then* $v \in sba_\top(\wp)(l)$.

Proof. Because $\forall \nu \in Val : sba([\nu/\alpha]\wp)(l) \subseteq sba_\top(\wp)(l)$ and by Theorem 2. The set-inclusion is obvious because sba and $sba_\top$ use the same monotonic rules to close the constraints and both of the two initial constraint sets are identical except that $sba([\nu/\alpha]\wp)$ has $\mathcal{X}_\alpha \supseteq \nu$ whereas $sba_\top(\wp)$ has a larger one: $\mathcal{X}_\alpha \supseteq \top$. □

3.3.2 Dynamic Analysis: Estimating Variants

In order for the analysis to approximate input-dependent values of expressions, we have to defer the solving phase to run-time. At run-time, we can replace the $\top$ (universe) in the input constraint $\mathcal{X}_\alpha \supseteq \top$ by the actual input ν:

$$\begin{array}{lll} \text{let} & \wp \rhd \mathcal{C} & \text{set-up constraints at compile-time} \\ \text{in} & S^*(\mathcal{C} \cup \{\mathcal{X}_\alpha \supseteq \nu\}) & \text{solve constraints at run-time} \end{array}$$

Above process estimates the expression values specific to input ν.

Definition 3 **(sba_ν)** *Let $\wp$ be a term with a free variable α and let $\wp \rhd \mathcal{C}$. $sba_\nu(\wp)(l) \stackrel{def}{=} [\![atom(S^*(\mathcal{C} \cup \{\mathcal{X}_\alpha \supseteq \nu\})(\mathcal{X}_l))]\!]$.*

Theorem 4 (Safety of sba_ν) *If $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ then $v \in sba_\nu(\wp)(l)$.*

Proof. Because $sba_\nu(\wp) = sba([\nu/\alpha]\wp)$ and by Theorem 2. $\square$

This dynamic analysis is the starting point for the following sections. We will present how to transform this dynamic analysis so that the transformed analysis should safely and quickly finish at the analyzed program's input occurrence.

4 Derivative I: Deferring Dynamic Constraints

Much of computing the dynamic analysis $S^*(\mathcal{C} \cup \{\mathcal{X}_\alpha \supseteq \nu\})$ at the analyzed program's run-time can be done at its compile-time. Even though the constraint $\mathcal{X}_\alpha \supseteq \nu$ is available only at the program's run-time, we can solve at its compile-time input-independent constraints among $\mathcal{C}$, while we defer input-dependent ones to run-time.

Program's input-dependent constraints $\mathcal{X} \supseteq se_\square$ have the same syntax as the normal constraints except that the set expressions $se_\square$ are marked input-dependent (subscript $\square$). Input-dependent set expressions $se_\square$ are those that cannot be dissolved into atomic ones at compile-time because they depend on the input. Set-variable $\mathcal{X}_\alpha$ for the input is input-dependent. Non-atomic compound expressions are input-dependent if one of their sub-parts is. Hence there are four kinds of input-dependent set expressions:

$$se_\square ::= \mathcal{X}_\alpha \mid \kappa_\square^{-1}\,\mathcal{X} \mid app_\square(\mathcal{X}_1, \mathcal{X}_2) \mid case_\square(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3).$$

Identifying input-dependent constraints is simple. When we apply the solving rules S to constraints, we also propagate the input-dependencies, starting from the set constraints $\mathcal{X} \supseteq \mathcal{X}_\alpha$ that have the input set-variable on their right-hand-sides. Note that the solving rules in S dissolve a compound set-constraints into smaller ones if given a firing constraint. Thus when the firing constraint is input-dependent so is the compound set-constraint. For example, consider

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq \mathcal{X}_e \quad \mathcal{X}_x \supseteq \mathcal{X}_2}.$$

The firing constraint $\mathcal{X}_1 \supseteq \lambda x.e$ allows $\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2)$ to dissolve. If this firing constraint is input-dependent, we mark the compound set expression $app(\mathcal{X}_1, \mathcal{X}_2)$ as input-dependent:

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq se_\square}{\mathcal{X} \supseteq app_\square(\mathcal{X}_1, \mathcal{X}_2)}.$$

Other cases for propagating input-dependencies are similarly defined.

Figure 5 has the rules D for identifying input-dependent constraints. Each rule is juxtaposed with its corresponding rule of S. The last rule propagates the input-dependencies across the single-variable set expression, which is neither compound nor yet completely dissolved.

The whole process is defined as:

let	$\wp \triangleright \mathcal{C}$	set-up constraints at compile-time
	$\mathcal{C}' = (S \cup D)^*(\mathcal{C})$	partially solve (S) and defer (D) at compile-time
in	$S^*_\square(\lvert\mathcal{C}'\rvert \cup \{\mathcal{X}_\alpha \supseteq \nu\})$	solve deferred constraints at run-time

We collect ($\wp \triangleright \mathcal{C}$) the initial constraint set $\mathcal{C}$ from program $\wp$ (rules in Figure 3). We apply the rules S to the $\mathcal{C}$ for solving input-independent constraints and at the same time the rules D for identifying and deferring input-dependent constraints. These two phases can be done at compile-time. The result constraint set $\mathcal{C}'$, which is partially solved, is finally solved at run-time with input ν. At run-time we apply rules $S_\square$, which are the same as the rules S except that they handle the compound set expressions marked with $\square$. Constraint set $|\mathcal{C}'|$ is the set of solved or deferred constraints in $\mathcal{C}'$:

$$|\mathcal{C}'| = \{\mathcal{X} \supseteq ae \mid \mathcal{X} \supseteq ae \in \mathcal{C}'\} \cup \{\mathcal{X} \supseteq se_\square \mid \mathcal{X} \supseteq se_\square \in \mathcal{C}'\}.$$

Definition 4 (Deferred analysis $dsba$**)** *Let $\wp$ be a term with a free variable α, $\wp \triangleright \mathcal{C}$, and ν be an input value.* $dsba_\nu(\wp)(l) \stackrel{def}{=} [\![atom(\mathcal{C}''(\mathcal{X}_l))]\!]$, *where* $\mathcal{C}' = (S \cup D)^*(\mathcal{C})$ *and* $\mathcal{C}'' = S^*_\square(|\mathcal{C}'| \cup \{\mathcal{X}_\alpha \supseteq \nu\})$.

Theorem 5 (Safety of $dsba$**)** *If* $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ *then* $v \in dsba_\nu(\wp)(l)$.

Proof. This is because $dsba_\nu(\wp) = sba_\nu(\wp)$. The only difference between the two processes is that $dsba_\nu(\wp)$ defers input-dependent constraints and solves them with run-time input ν. Here, the run-time solving rules are identical to the rules in $sba_\nu(\wp)$ and the semantics of the deferred constraints are the same as before. Hence the final result from the deferred set-based analysis should be the same as the one from $sba_\nu(\wp)$. $\square$

Note that the run-time process $S^*_\square(|\mathcal{C}'| \cup \{\mathcal{X}_\alpha \supseteq \nu\})$ of solving the deferred constraints $|\mathcal{C}'|$ has some delays. Because each deferred constraint's ($\mathcal{X} \supseteq se_\square$ in $|\mathcal{C}'|$) firing constraint is a solved atomic one about its sub-part, it has to wait until its sub-part is solved.

Constraint transformation to avoid this delay is our next step.

5 Derivative II: Preparing Dynamic Constraints via Static Value Slicing

Solving each deferred constraint $\mathcal{X} \supseteq se_\square$ can be short-cut if its firing constraint is not about a sub-part of $se_\square$ but about the program's input. For example, consider a resolution rule for $\mathcal{X} \supseteq case_\square(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)$:

$$\frac{\mathcal{X} \supseteq case_\square(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa\,\mathcal{Y}}{\mathcal{X} \supseteq \mathcal{X}_2}$$

Because the firing constraint $\mathcal{X}_1 \supseteq \kappa\,\mathcal{Y}$ is about the pivoting expression ($\mathcal{X}_1$) we have to wait until the constraint on $\mathcal{X}_1$ is dissolved to have $\kappa\,\mathcal{Y}$. This delay is removed if our transformed rule is: if the program's input is included in V the case expression has the values $\mathcal{X} \supseteq \mathcal{X}_2$. The condition for V must be that every input in V to the program makes e_1 (expression for $\mathcal{X}_1$) evaluate into values with constructor κ.

A static analysis named "static value-slicing" estimates such set V. Static value-slicing is a compile-time analysis to answer the question: in order for an expression's value to be included in a set V, what will be a necessary condition for the expression's environment?

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e_3}{\mathcal{X} \supseteq \mathcal{X}_3 \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa\, \mathcal{Y}}{\mathcal{X} \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa'\, \mathcal{Y} \quad \kappa' \neq \kappa}{\mathcal{X} \supseteq \mathcal{X}_3}$$

$$\frac{\mathcal{X} \supseteq \kappa^{-1}\, \mathcal{Y} \quad \mathcal{Y} \supseteq \kappa\, \mathcal{Z}}{\mathcal{X} \supseteq \mathcal{Z}}$$

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

Solving rules S (Figure 4)
at compile-time

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq se_\square}{\mathcal{X} \supseteq app_\square(\mathcal{X}_1, \mathcal{X}_2)}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq se_\square}{\mathcal{X} \supseteq case_\square(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)}$$

$$\frac{\mathcal{X} \supseteq \kappa^{-1}\, \mathcal{Y} \quad \mathcal{Y} \supseteq se_\square}{\mathcal{X} \supseteq \kappa_\square^{-1}\, \mathcal{Y}}$$

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq se_\square}{\mathcal{X} \supseteq se_\square}$$

Deferring rules D
at compile-time

Figure 5: Rules D for identifying and deferring input-dependent constraints

5.1 Static Value Slicing

Conventionally, static slicing[Tip94] is a compile-time technique to select program parts (syntactic objects like expressions or variables) that are related to a selected expression.

Static *value*-slicing is a static slicing where we slice expressions' values, not expressions. The slice criterion $e \subseteq V$ is a condition that expression e's values must be included in V. The slice result

$$\mathrm{SVS}_\wp(e \subseteq V)$$

is a map from the expressions of $\wp$ to their value sets that are *necessary* for the program to compute e's values in V. To be precise, for every input value in the slice result $\mathrm{SVS}_\wp(e \subseteq V)(\alpha)$, if it makes the program compute e's values then the values are in V.

Example 1 Consider the following program $\wp$:

```
datatype t = A | B | C
case x of
     A => B
   | _ => x
```

If the input `x` is `A` or `B`, the program returns `B`. If `x` is `C`, it returns `C`. Static value-slicing $\mathrm{SVS}_\wp(\wp \subseteq \{\texttt{B}\})(\texttt{x})$ for slice criterion $\wp \subseteq \{\texttt{B}\}$ must be a subset of $\{\texttt{A},\texttt{B}\}$. Similarly, for criterion $\wp \subseteq \{\texttt{C}\}$, $\mathrm{SVS}_\wp(\wp \subseteq \{\texttt{C}\})(\texttt{x}) \subseteq \{\texttt{C}\}$. □

We present one such static value-slicing analysis as a set constraint system. Set constraints are of the form

$$se_l \subseteq se_r$$

indicating that set se_l has to be included in set se_r. Figure 7 has the definitions of the constraints, three alternative rules for deriving initial slicing constraints, and slicing propagation rules B.[3] The rules B are straightforward from the semantics of constraints. Driving rules ($\rhd_i$) for initial slicing constraints are also intuitive:

- If $(\texttt{con}\ \kappa\ e_1)_l$ is sliced to $\mathcal{X}_l$, we have to slice the sub-expression e_1: $\kappa\,\mathcal{X}_1 \subseteq \mathcal{X}_l$. Similarly for the `decon` case.
- If application expression $(e_1\ e_2)_l$ is sliced to $\mathcal{X}_l$, there are three value sets we have to slice: e_1's, e_2's, and the called function's body expression's. The function expression e_1 has to be sliced to the set $\mathit{Lam}_\wp(e_1)$ of functions that can be called. For each function $\lambda x.e_3 \in \mathit{Lam}_\wp(e_1)$, the argument expression e_2 has to be sliced to the values of x, and the called function's body has to be sliced to $\mathcal{X}_l$.

  The $\mathit{Lam}_\wp$ table maps expression e to a safe approximation of functions that the expression may have during program $\wp$'s executions for all inputs. The rules L in Figure 6 determines the table:

  $$\mathit{Lam}_\wp(e) = \{\lambda x.e' \mid e \to \lambda x.e' \in L^*_\wp(\text{exprs of } \wp)\}.$$

  Relation $e \to \lambda x.e'$ indicates that e may evaluate into function $\lambda x.e'$. Safety of the $\mathit{Lam}_\wp$ table is obvious.
- If case expression $(\texttt{case}\ e_1\ \kappa\ e_2\ e_3)_l$ is sliced to $\mathcal{X}_l$, there are three *alternative* slicings, depending on the three possible execution flows: only the first branch is taken, only the second branch is taken, or both branches are taken. Each situation requires different slicing rules. For the first situation, we slice e_1 to $\kappa\top$ ($\top$ for universe) and e_2 to $\mathcal{X}_l$ (`case` rule in $\rhd_1$). For the second situation, we slice e_1 to $\overline{\kappa}\,\top$ (values not constructed with κ) and e_3 to $\mathcal{X}_l$ (`case` rule in $\rhd_2$). For the third situation, we should not slice e_1 but slice both expressions e_2 and e_3 to $\mathcal{X}_l$ (`case` rule in $\rhd_3$).

The process of static value-slicing $\mathrm{SVS}_\wp(e \subseteq V)$ is defined as: [4]

let	$\wp \rhd_i \mathcal{C}_i$, for $i = 1,2,3$	set-up three initial slicing-constraint sets
in	$\bigvee_{i=1}^{3} B^*(\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\})$	disjuction of three slicing results from the criterion

The set of atomic slicing-constraints closed by the rules B is the greatest model (gm) of $\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$. The greatest model always exists because our constraints are co-definite [CP98].

Theorem 6 *Let $\wp$ be a term with one free variable α, e be a sub-expression of $\wp$, V be a set of values, and $\wp \rhd_i \mathcal{C}_i$. Then $[\![\mathit{atom}(B^*(\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}))]\!]$ is the greatest model of $\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$.*

Proof. Let $\mathcal{C}' = \mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$. We first prove $gm(\mathcal{C}') = gm(B^*(\mathcal{C}'))$ by showing that B always adds constraints that preserve the $gm(\mathcal{C}')$. Then we prove $gm(B^*(\mathcal{C}')) = gm(\mathit{atom}(B^*(\mathcal{C}')))$. Details are in Appendix A.1. $\square$

The atomic slicing-constraints are slicing constraints $\mathcal{X} \subseteq ae$ whose set-expression ae is atomic.

Static value-slicing is the set of the greatest models for three slicing constraints:

Definition 5 (Static Value-Slicing $\mathbf{SVS}_\wp(e \subseteq V)$) *Let $\wp$ be a term with one free variable α, e be a sub-expression of $\wp$, and $\wp \rhd_i \mathcal{C}_i$ for i = 1,2,3.*

$$\mathit{SVS}_\wp(e \subseteq V) \stackrel{def}{=} \{[\![\mathit{atom}(B^*(\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}))]\!] \mid i = 1, 2, 3\}.$$

[3] "B" because the slicing propagations are roughly "backward."

[4] Note that for n case expressions in the program, we can start from 3^n sets of initial slicing constraints (3 slicings for each case expression), which we avoid for the slicing's cost-accuracy balance.

$$\frac{\lambda x.e \in \wp}{\lambda x.e \rightarrow \lambda x.e}$$

$$\frac{e_1 \rightarrow \lambda x.e_1' \quad e_2 \rightarrow \lambda y.e_2' \quad e_1\ e_2 \in \wp}{x \rightarrow \lambda y.e_2'}$$

$$\frac{e_1 \rightarrow \lambda x.e_1' \quad e_1' \rightarrow \lambda y.e_2'}{e_1\ e_2 \rightarrow \lambda y.e_2'} \qquad \frac{\lambda x.e \in \wp}{\texttt{decon}\ \kappa\ e_1 \rightarrow \lambda x.e}$$

$$\frac{e_2 \rightarrow \lambda x.e}{\texttt{case}\ e_1\ \kappa\ e_2\ e_3 \rightarrow \lambda x.e} \qquad \frac{e_3 \rightarrow \lambda x.e}{\texttt{case}\ e_1\ \kappa\ e_2\ e_3 \rightarrow \lambda x.e}$$

Figure 6: Safe call-graph estimation rules $L_\wp$

In the following sections, we will abuse the notation $\mathrm{SVS}_\wp(e \subseteq V)(\alpha)$ to mean $\{v | v \in \Sigma(\alpha), \Sigma \in \mathrm{SVS}_\wp(e \subseteq V)\}$.

The value-slicing satisfies the condition that we anticipate:

Theorem 7 (Correctness of SVS$_\wp(e \subseteq V)$) *Let $\wp$ be a term with one free variable α, and e be a sub-expression of $\wp$. If $\Sigma \in SVS_\wp(e \subseteq V)$ then $\forall \nu \in \Sigma(\alpha) : ([\nu/\alpha]\wp \overset{*}{\longmapsto} \mathcal{E}[v_e] \Rightarrow v \in V)$.*

Proof. We use the set-based operational semantics[Hei92, Hei93] $\Sigma \vdash e \leadsto v$ as an intermediate step. First, we prove that $\Sigma \vdash e \leadsto v \Rightarrow v \in \Sigma(e)$. Furthermore Σ is a safe set environment with respect to $[\nu/\alpha]\wp$ and $\Sigma(e) \subseteq V$. Therefore, $[\nu/\alpha]\wp \overset{*}{\longmapsto} \mathcal{E}[v_e]$ implies $\Sigma \vdash e \leadsto v$, hence implies $v \in \Sigma(e)$, and hence implies $v \in V$. Details are in Appendix A.1. $\Box$

5.2 Short-cutting Dynamic Constraints

By using the static value-slicing techniques, an input-dependent constraint dissolves by a simple membership test on the program's input. By this preparation, when the program's input occurs at run-time each deferred yet prepared dynamic constraint can immediately and simultaneously start to resolve.

Consider the dynamic constraint for the case expression: $\mathcal{X} \supseteq case_\Box(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)$. Its firing constraint is either $\mathcal{X}_1 \supseteq \kappa\, \mathcal{Y}$ or $\mathcal{X}_1 \supseteq \kappa'\, \mathcal{Y}$ ($\kappa' \neq \kappa$). Suppose that the inputs in set V_1 (resp., V_2) make the pivoting expression $\mathcal{X}_1$ have data made with κ (resp., κ'). Such sets are the results of the static value-slicings:

$$\begin{array}{rcl} V_1 & = & \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \kappa\, \top)(\alpha) \\ V_2 & = & \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \overline{\kappa}\, \top)(\alpha). \end{array}$$

Using these two sets, the dynamic constraint $\mathcal{X} \supseteq case_\Box(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)$ is re-formulated into three constraints

$$\mathcal{X} \supseteq if_\Box(V_1, \mathcal{X}_2), \mathcal{X} \supseteq if_\Box(V_2, \mathcal{X}_3), \text{ and } \mathcal{X} \supseteq if_\Box(\overline{V_1 \cup V_2}, \mathcal{X}_2 \cup \mathcal{X}_3)$$

where the set expression $if_\Box(V, \mathcal{X})$ indicates the set $\mathcal{X}$ only when the program input is an element of V. The last $if_\Box$ constraint is for when the two exclusive sets V_1 and V_2 can be non-exhaustive because they are "necessary" sets for the slice criterion. It covers the case when the input is included neither of the two sets.

Slicing constraint is $se_l \subseteq se_r$.

Syntax of slicing set-expressions:

$$\begin{array}{rcll} se_l & ::= & \mathcal{X}_i & \text{set variables} \\ & \mid & \lambda x.e & \text{lambdas} \\ & \mid & \kappa\,\mathcal{X}_i & \text{data sets} \end{array} \qquad \begin{array}{rcll} se_r & ::= & \mathcal{X}_i & \text{set variables} \\ & \mid & \kappa\,\mathcal{X}_i & \text{data sets} \\ & \mid & \kappa\,\top & \text{data sets} \\ & \mid & \overline{\kappa}\,\top & \text{data sets} \\ & \mid & Lam_{\wp}(e) & \text{lambda sets} \\ & \mid & \top & \text{universe set} \end{array}$$

$$\begin{array}{rcll} i & \in & \{l, x \mid e_l \in Expr, x \in Var\} & \text{indices for the sub-expressions and variables} \\ ae & ::= & \kappa\,\mathcal{X}_i \mid \kappa\,\top \mid \overline{\kappa}\,\top \mid \top \mid Lam_{\wp}(e) & \text{atomic set expression} \end{array}$$

Semantics of slicing set-expressions:

$$\begin{array}{rcl} \mathcal{I}(\mathcal{X}_i) & \subseteq & Val \\ \mathcal{I}(\top) & = & Val \\ \mathcal{I}(Lam_{\wp}(e)) & = & Lam_{\wp}(e) \\ \mathcal{I}(\kappa\,\top) & = & \{\kappa(v) \mid v \in Val\} \end{array} \qquad \begin{array}{rcl} & & \\ \mathcal{I}(\lambda x.e) & = & \{\lambda x.e\} \\ \mathcal{I}(\kappa\,\mathcal{X}) & = & \{\kappa(v) \mid v \in \mathcal{I}(\mathcal{X})\} \\ \mathcal{I}(\overline{\kappa}\,\top) & = & \{\kappa'(v) \mid \kappa' \neq \kappa, v \in Val\} \end{array}$$

Rules $\rhd_1$, $\rhd_2$, and $\rhd_3$ for deriving three alternative sets of initial slicing constraints:

$$x_l \rhd_i \{\,\mathcal{X}_x \subseteq \mathcal{X}_l\} \qquad \frac{e_1 \rhd_i \mathcal{C}_1}{(\lambda x.e_1)_l \rhd_i \{\lambda x.e_1 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1}$$

$$\frac{e_1 \rhd_i \mathcal{C}_1}{(\texttt{con}\ \kappa\ e_1)_l \rhd_i \{\kappa\,\mathcal{X}_1 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1} \qquad \frac{e_1 \rhd_i \mathcal{C}_1}{(\texttt{decon}\ \kappa\ e_1)_l \rhd_i \{\mathcal{X}_1 \subseteq \kappa\,\mathcal{X}_l\} \cup \mathcal{C}_1}$$

$$\frac{e_1 \rhd_i \mathcal{C}_1 \quad e_2 \rhd_i \mathcal{C}_2 \quad Lam_{\wp}(e_1) \ni \lambda x.e_3}{(e_1\ e_2)_l \rhd_i \{\mathcal{X}_1 \subseteq Lam_{\wp}(e_1), \mathcal{X}_2 \subseteq \mathcal{X}_x, \mathcal{X}_3 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{e_1 \rhd_1 \mathcal{C}_1 \quad e_2 \rhd_1 \mathcal{C}_2 \quad e_3 \rhd_1 \mathcal{C}_3}{(\texttt{case}\ e_1\ \kappa\ e_2\ e_3)_l \rhd_1 \{\mathcal{X}_1 \subseteq \kappa\,\top, \mathcal{X}_2 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\frac{e_1 \rhd_2 \mathcal{C}_1 \quad e_2 \rhd_2 \mathcal{C}_2 \quad e_3 \rhd_2 \mathcal{C}_3}{(\texttt{case}\ e_1\ \kappa\ e_2\ e_3)_l \rhd_2 \{\mathcal{X}_1 \subseteq \overline{\kappa}\,\top, \mathcal{X}_3 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\frac{e_1 \rhd_3 \mathcal{C}_1 \quad e_2 \rhd_3 \mathcal{C}_2 \quad e_3 \rhd_3 \mathcal{C}_3}{(\texttt{case}\ e_1\ \kappa\ e_2\ e_3)_l \rhd_3 \{\mathcal{X}_2 \subseteq \mathcal{X}_l, \mathcal{X}_3 \subseteq \mathcal{X}_l\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

Rules B for slicing propagation:

$$\frac{\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq ae}{\mathcal{X} \subseteq ae} \qquad \frac{\kappa\,\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \kappa\,\mathcal{Z}}{\mathcal{X} \subseteq \mathcal{Z}} \qquad \frac{\kappa\,\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \kappa\,\top}{\mathcal{X} \subseteq \top}$$

$$\frac{\kappa\,\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \overline{\kappa'}\,\top \quad \kappa' \neq \kappa}{\mathcal{X} \subseteq \top} \qquad \frac{\kappa\,\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \top}{\mathcal{X} \subseteq \top}$$

Figure 7: Set constraints for static value-slicing: syntax, semantics and slicing rules

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e_3}{\mathcal{X} \supseteq \mathcal{X}_3 \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa\, \mathcal{Y}}{\mathcal{X} \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq \kappa'\, \mathcal{Y} \quad \kappa' \neq \kappa}{\mathcal{X} \supseteq \mathcal{X}_3}$$

$$\frac{\mathcal{X} \supseteq \kappa^{-1}\, \mathcal{Y} \quad \mathcal{Y} \supseteq \kappa\, \mathcal{Z}}{\mathcal{X} \supseteq \mathcal{Z}}$$

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

Solving rules S (Figure 4)
at compile-time

$$\frac{\mathcal{X} \supseteq app(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq se_\Box}{\mathcal{X} \supseteq app_\Box(\mathcal{X}_1, \mathcal{X}_2)}$$

$$\frac{\begin{array}{c}\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq se_\Box \\ V_1 = \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \kappa\, \top)(\alpha) \\ V_2 = \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \overline{\kappa}\, \top)(\alpha)\end{array}}{\begin{array}{c}\mathcal{X} \supseteq if_\Box(V_1, \mathcal{X}_2) \quad \mathcal{X} \supseteq if_\Box(V_2, \mathcal{X}_3) \\ \mathcal{X} \supseteq if_\Box(\overline{V_1 \cup V_2}, \mathcal{X}_2 \cup \mathcal{X}_3)\end{array}}$$

$$\frac{\mathcal{X} \supseteq \kappa^{-1}\, \mathcal{Y} \quad \mathcal{Y} \supseteq se_\Box}{\mathcal{X} \supseteq \kappa_\Box^{-1}\mathcal{Y}}$$

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq se_\Box}{\mathcal{X} \supseteq se_\Box}$$

Preparing and deferring rules P
at compile-time

$$\frac{\mathcal{X} \supseteq if_\Box(S, \mathcal{X}_1) \quad \text{input } \nu \in S}{\mathcal{X} \supseteq \mathcal{X}_1} \qquad \frac{\mathcal{X} \supseteq app_\Box(\mathcal{X}_1, \mathcal{X}_2) \quad \mathcal{X}_1 \supseteq \lambda x.e}{\mathcal{X} \supseteq \mathcal{X}_e \quad \mathcal{X}_x \supseteq \mathcal{X}_2}$$

$$\frac{\mathcal{X} \supseteq \kappa_\Box^{-1}\, \mathcal{Y} \quad \mathcal{Y} \supseteq \kappa\, \mathcal{Z}}{\mathcal{X} \supseteq \mathcal{Z}} \qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \quad \mathcal{Y} \supseteq ae}{\mathcal{X} \supseteq ae}$$

Solving rules S_P at run-time

Figure 8: Preparing rules P at compile-time and solving rules S_P at run-time

The firing constraint for $\mathcal{X} \supseteq if_\Box(V, \mathcal{X})$ is a simple test $\nu \in V$ for the program's input ν:

$$\frac{\mathcal{X} \supseteq if_\Box(V, \mathcal{X}_1) \quad \nu \in V}{\mathcal{X} \supseteq \mathcal{X}_1}$$

The membership test $\nu \in V$ for regular tree grammar V takes polynomial time [CDG+99]: the size of ν $\times$ the number of non-terminals in V $\times$ the sum of the function symbols' arities (in our case, data constructors' arties).

The rule for defining an input-dependent constraint for the `case` expression is changed as:

$$\frac{\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq se_\Box}{\mathcal{X} \supseteq case_\Box(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3)} \quad \text{becomes} \quad \frac{\begin{array}{c}\mathcal{X} \supseteq case(\mathcal{X}_1, \kappa, \mathcal{X}_2, \mathcal{X}_3) \quad \mathcal{X}_1 \supseteq se_\Box \\ V_1 = \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \kappa\, \top)(\alpha) \\ V_2 = \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \overline{\kappa}\, \top)(\alpha)\end{array}}{\begin{array}{c}\mathcal{X} \supseteq if_\Box(V_1, \mathcal{X}_2) \quad \mathcal{X} \supseteq if_\Box(V_2, \mathcal{X}_3) \\ \mathcal{X} \supseteq if_\Box(\overline{V_1 \cup V_2}, \mathcal{X}_2 \cup \mathcal{X}_3)\end{array}}$$

The rules P for identifying, preparing, and deferring input-dependent constraints are in Figure 8, together with the rules S_p for solving the prepared constraints at run-time.

```
℘ =
(λpacket.
  (λethertype. λip. λarp.
    case ethertype of
      IP => (λipsrc. λipdst. case_a [ipsrc] of FOO => TRUE              ······ (*)
                                          | _ => e
            ) #1(ip) #2(ip)
    | _ => ···
  ) #1(packet) #1(#2(packet)) #2(#2(packet))
) α
```

Figure 9: Example packet filter program

The whole process of preparing the analysis is defined as:

let	$\wp \triangleright \mathcal{C}$	set-up constraints at compile-time
	$\mathcal{C}' = (S \cup P)^*(\mathcal{C})$	partially solve, prepare at compile-time
in	$S_P^*(\lvert\mathcal{C}'\rvert \cup \{\mathcal{X}_\alpha \supseteq \nu\})$	solve prepared constraints at run-time

We collect program's initial constraints $\mathcal{C}$ using the $\triangleright$ rules of Figure 3. Apply the rules S to the $\mathcal{C}$ for solving input-independent constraints and at the same time the rules P for identifying and preparing input-dependent constraints. The result constraint set $\mathcal{C}'$, which is partially solved and prepared for quick convergence, is finally solved at run-time with the program input ν. As before, $|\mathcal{C}'|$ is the set of solved or deferred constraints in $\mathcal{C}'$.

Definition 6 (Prepared analysis *psba***)** *Let $\wp$ be a term with a free variable α, $\wp \triangleright \mathcal{C}$, and ν be an input value. $psba_\nu(\wp)(l) \stackrel{def}{=} [\![atom(\mathcal{C}''(\mathcal{X}_l))]\!]$, where $\mathcal{C}' = (S \cup P)^*(\mathcal{C})$ and $\mathcal{C}'' = S_P^*(|\mathcal{C}'| \cup \{\mathcal{X}_\alpha \supseteq \nu\})$.*

The prepared analysis safely estimates the input-dependent properties:

Theorem 8 (Safety of *psba***)** *If $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ then $v \in psba_\nu(\wp)(l)$.*

Proof. We prove the safety of *psba* by showing that *psba* is a safe approximation of sba_ν. We define two continuous functions $\mathcal{F}_\nu$ and $\mathcal{P}_\nu$ that correspond [CC95] to the closure operations $sba_\nu(\wp)$ and $psba_\nu(\wp)$, respectively. Then, we prove by the fixpoint induction that the least fixpoint of $\mathcal{P}_\nu$ is larger than or equal to the least fixpoint of $\mathcal{F}_\nu$. Details of this proof is in Appendix A.2. □

6 Example

Let's consider the packet filter program (Figure 9), which is used in [MJ93]. This program gets a packet whose data structure is

$$\begin{array}{rcl} packet & ::= & \langle ethertype, \langle ip, arp\rangle\rangle \\ ip & ::= & \langle ipsrc, ipdst\rangle \\ arp & ::= & \langle arpsrc, arpdst\rangle \end{array}$$

and returns TRUE if the input packet's ip source is FOO. [5]

Consider the analysis for the $\mathtt{case}_a$ expression (line marked with $(*)$). The initial set constraint (from $\wp \rhd \mathcal{C}$) for the $\mathtt{case}_a$ expression is

$$\mathcal{X}_{\mathtt{case}_a} \supseteq \mathit{case}(\mathcal{X}_{\mathtt{ipsrc}}, \mathtt{FOO}, \mathcal{X}_{\mathtt{TRUE}}, \mathcal{X}_e).$$

Note that $\mathcal{X}_{\mathtt{ipsrc}}$ is input-dependent because the variable ipsrc is bound to a sub-structure of the input α.

We short-cut its firing constraints ($\mathcal{X}_{\mathtt{ipsrc}} \supseteq \mathtt{FOO}\,\mathcal{Y}$ and $\mathcal{X}_{\mathtt{ipsrc}} \supseteq \kappa\,\mathcal{Y}$ for $\kappa \neq \mathtt{FOO}$) so that they become membership tests on the program's input. Such sets for the inclusion tests are the results from the static value-slicings:

$$\mathrm{SVS}_\wp(\mathcal{X}_{\mathtt{ipsrc}} \subseteq \mathtt{FOO}\,\top) \quad \text{and} \quad \mathrm{SVS}_\wp(\mathcal{X}_{\mathtt{ipsrc}} \subseteq \overline{\mathtt{FOO}}\,\top).$$

Figure 10 shows some snapshots during $\mathrm{SVS}_\wp(\mathit{ipsrc} \subseteq \mathtt{FOO}\,\top)(\alpha)$. Its result for the input $\mathcal{X}_\alpha$ is

$$\mathcal{X}_\alpha \subseteq \langle \mathtt{IP}, \langle\langle \mathtt{FOO}, \mathtt{FOO}\rangle, \top\rangle\rangle \quad \text{or} \quad \mathcal{X}_\alpha \subseteq \langle \top, \langle\langle \mathtt{FOO}, \top\rangle, \top\rangle\rangle.$$

Therefore, we replace the firing constraint $\mathcal{X}_{\mathtt{ipsrc}} \supseteq \mathtt{FOO}\,\mathcal{Y}$ by the membership test $\alpha \in \langle \top, \langle\langle \mathtt{FOO}, \top\rangle, \top\rangle\rangle$ for the program input α. That is, at run-time, if the input passes the above inclusion test, the prepared constraint

$$\mathcal{X}_{\mathtt{case}_a} \supseteq \mathit{if}_\Box(\langle \top, \langle\langle \mathtt{FOO}, \top\rangle, \top\rangle\rangle, \mathcal{X}_{\mathtt{TRUE}})$$

is resolved into

$$\mathcal{X}_{\mathtt{case}_a} \supseteq \mathcal{X}_{\mathtt{TRUE}}.$$

7 Related Works

Other works on specializing data-flow analyses at run-time have three limitations in comparison with our technique: the program's control flow has to be known beforehand, a static preparation for dynamic acceleration (via static value-slicing) is missing, the correctness of the specialization is not formally defined and proved. Sharma *et al.*'s technique [SAS98] identifies program points whose analyses have to conservatively subsume multiple data flows. They defer such program points' analyses to run-time. At run-time, the multiple data flow edges into each deferred program point is resolved into a single one. Their technique assumes that a program's control flow graph is known, hence cannot be applied to programs with first-class functions. The run-time specialization has to wait for the running program's control to reach at the deferred program point. No particular preparation is done to reduce this delay. Moon *et al.*'s dynamic dependence analysis [MHM98] for Fortran programs has the same limitations.

Ammons and Larus use profile data to improve the data-flow analysis precision [AL98]. The analysis accuracy is improved by isolating and focusing the data-flow analysis on the hot paths of the program's control flow graph. Their technique has the overhead of collecting profile data, identifying hot-paths, isolating the data-flow analysis for hot-paths, and reconstructing a safe analysis for the whole program flow. Our technique has no such an overhead other than the cost for the flow-analysis itself.

[5] Note that the example program has pairs and selections, whose analyses rules are omitted for brevity in this paper. We must be careful that additional constraints are all co-definite. We have to add the slicing propagation rule:

$$\frac{\mathcal{X} \subseteq \mathtt{\#}i\,\mathcal{Y} \quad \mathcal{Y} \subseteq \langle \mathcal{Z}_1, \mathcal{Z}_2\rangle}{\mathcal{X} \subseteq \mathcal{Z}_i}$$

The three sets of constraints of $\mathrm{SVS}_\wp(\mathcal{X}_{\mathtt{is}} \subseteq \mathtt{FOO})(\alpha)$			
from $\mathcal{C}_1$	from $\mathcal{C}_2$	from $\mathcal{C}_3$	
$\mathcal{X}_{\mathtt{et}} \subseteq \mathtt{IP}$ $\mathcal{X}_{\mathtt{is}} \subseteq \mathtt{FOO}$	$\mathcal{X}_{\mathtt{et}} \subseteq \overline{\mathtt{IP}}$ $\mathcal{X}_{\mathtt{is}} \subseteq \overline{\mathtt{FOO}}$		initial constraints from the $\triangleright_i$ rules
$\mathtt{\#1}\ \mathcal{X}_{\mathtt{i}} \subseteq \mathtt{FOO}$ $\mathtt{\#2}\ \mathcal{X}_{\mathtt{i}} \subseteq \mathtt{FOO}$ $\mathcal{X}_{\mathtt{\#1i}} \subseteq \langle \mathtt{FOO}, \top \rangle$ $\mathcal{X}_{\mathtt{\#2i}} \subseteq \langle \top, \mathtt{FOO} \rangle$ $\mathtt{\#1}\ \mathtt{\#2}\ \mathcal{X}_{\mathtt{p}} \subseteq \langle \mathtt{FOO}, \mathtt{FOO} \rangle$ $\mathtt{\#2}\ \mathcal{X}_{\mathtt{p}} \subseteq \langle \langle \mathtt{FOO}, \mathtt{FOO} \rangle, \top \rangle$ $\mathcal{X}_{\mathtt{p}} \subseteq \langle \top, \langle \langle \mathtt{FOO}, \mathtt{FOO} \rangle, \top \rangle \rangle$ $\mathtt{\#1}\ \mathcal{X}_{\mathtt{p}} \subseteq \mathtt{IP}$ $\mathcal{X}_{\mathtt{p}} \subseteq \langle \mathtt{IP}, \top \rangle$ $\mathcal{X}_{\mathtt{p}} \subseteq \langle \mathtt{IP}, \langle \langle \mathtt{FOO}, \mathtt{FOO} \rangle, \top \rangle \rangle$		$\mathtt{\#1}\ \mathcal{X}_{\mathtt{i}} \subseteq \mathtt{FOO}$ $\mathcal{X}_{\mathtt{\#1i}} \subseteq \langle \mathtt{FOO}, \top \rangle$ $\mathtt{\#1}\ \mathtt{\#2}\ \mathcal{X}_{\mathtt{p}} \subseteq \langle \mathtt{FOO}, \top \rangle$ $\mathtt{\#2}\ \mathcal{X}_{\mathtt{p}} \subseteq \langle \langle \mathtt{FOO}, \top \rangle, \top \rangle$ $\mathcal{X}_{\mathtt{p}} \subseteq \langle \top, \langle \langle \mathtt{FOO}, \top \rangle, \top \rangle \rangle$	propagated slicing constraints
$\mathcal{X}_\alpha \subseteq \langle \mathtt{IP}, \langle \langle \mathtt{FOO}, \mathtt{FOO} \rangle, \top \rangle \rangle$	no solution	$\mathcal{X}_\alpha \subseteq \langle \top, \langle \langle \mathtt{FOO}, \top \rangle, \top \rangle \rangle$	final result for α

where subscript	`p`	for	`packet`	in $\wp$
	`et`	for	`ethertype`	in $\wp$
	`i`	for	`ip`	in $\wp$
	`is`	for	`ipsrc`	in $\wp$

Figure 10: Snapshots of $\mathrm{SVS}_\wp(\mathcal{X}_{\mathtt{is}} \subseteq \mathtt{FOO})(\alpha)$

Run-time code-generation techniques [CN96, LL95, AmPC+96, GMP+] are focused on dynamic code optimizations, not on deriving dynamic properties incurred from an input. They do not analyze at run-time the program's dynamic properties for particular inputs. Their analyses are at compile-time: basically binding-time analyses[CD93, Con90] from the programmer's annotations about for which parts of the input the code has to be specialized. At run-time, when an input for the parts is known, the code specialization process (adapted from the partial evaluation[CD93] technique) generates an optimized code tailored to the partial input.

Flanagan and Felleisen's componential set-based analysis [FF97] can be used orthogonally with our technique. Their techniques can be adapted to make our method work for separate, componential preparations for modular run-time specializations. Their constraint simplification is a technique of taming the indefinitely-generated constraints during the solving process. This simplification is not needed in our case because the number of generated constraints are bounded by the program size. We have to adapt their technique if our set-based static analysis is in such a class.

Our static value-slicing can be viewed as a subtype or refinement type inference [FM90, FP91, CDG96] with user's type annotation (slice criterion). The difference is that in the subtype or refinement type system, the type hierarchy (subtype or refinement relation) is much smaller than our domain of regular tree grammars for subsets of values. Reps and Turnidge's context analysis [RT96] and Liu's dependence analysis [Liu98] do not support higher-order functions and use only a small, fixed lattice for regular tree grammars. Bourdoncle's Syntox system[Bou93] is a kind of value-slicing tool. He demonstrated its use in debugging Pascal programs. Their value space is limited to the set of the integer ranges, while we support the set of arbitrary algebraic data. Biswas' demand-driven set-based analysis [Bis97] identifies the sub-expressions that contribute to the final result of the outermost expression. His work slices expressions (dead codes), not expression's values.

8 Conclusion

We presented a technique of using static analysis for estimating program's input-dependent properties. A static analysis that is originally designed for estimating the input-independent properties of programs is transformed into one that can safely estimate the input-dependent properties at the programs' input occurrence. No profile is collected and no probing codes inside the running program are needed.

Our idea is to defer the finish of the static analysis to the program's run-time. By analyzing the static analysis, we identify the parts of the analysis that are sensitive to the program's inputs, hence need to be deferred to the program's run-time. Then by using the *static value-slicing*, we re-formulate some of the dynamic parts so that they are solved by simple membership tests for the program's input. This re-formulation accelerates the analysis; once the program's input occurs the prepared dynamic parts can immediately and simultaneously start to resolve.

Our technique's strengths are: 1) any input-specific property of the programs can be analyzed if there exists a corresponding set-based analysis, hence more general than the conventional profiling that usually estimates only execution frequencies, 2) it has no overhead on running programs (as in profiling) because it is an independent, off-line analysis, 3) it can handle higher-order languages, 4) it is formally defined and proved correct.

References

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, January 1995.

[AL98] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[AmPC+96] Jeol Auslander, matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1996.

[Bis97] Sandip K. Biswas. A demand-driven set-based analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1997.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[BMS98] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[Bou93] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[CC95] Patrick Cousot and Radhia Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Lecture Notes in Computer Science*, volume 939, pages 293–308.

Springer-Verlag, proceedings of the 7th international conference on computer-aided verification edition, 1995.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In the Twentieth ACM Symposium on Principles of Programming Languages, January 1993.

[CDG96] Mario Coppo, Ferruccio Damiani, and Paola Giannini. Refinement types for program analysis. In *Lecture Notes in Computer Science*, volume 1145. Springer-Verlag, 1996.

[CDG+99] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, April 1999.

[CN96] Charles Consel and François Noël. A general approach for run-time specializationand its application to c. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[Con90] Charles Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the SIGPLAN Conference on Lisp and Functional Programming*, 1990.

[CP98] Witold Charatonik and Andreas Podelski. Co-definite set constraints. In *Lecture Notes in Computer Science*, volume 1379, pages 211–225. Springer-Verlag, Proceedings of the 9th International Conference on Rewriting Techniques and Applications - RTA'98 edition, 1998.

[Fea87] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Tranformation*, pages 165–195. Elsevier Science Publishers, 1987.

[FF97] Cormac Flangan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[GMP+] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in c. In *Proceedings of The ACM Symposium on Partial Evaluation and Program Manipulations*.

[Hei92] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[Hei93] Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, July 1993.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, June 1997.

[Liu98] Yanhong A. Liu. Dependence analysis for recursive data. In *Proceedings of the IEEE International Conference on Computer Language*, pages 206–215, May 1998.

[LL95] Mark Leone and Peter Lee. Optimizing ml with run-time code generation. Technical Report CMU-CS-95-205, Carnegie Mellon University, December 1995.

[MHM98] Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the ACM International Conference on Supercomputing*, July 1998.

[MJ93] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[RT96] Thomas Reps and Todd Turnidge. Program specialization via program slicing. In *Lecture Notes in Computer Science*, volume 1110, pages 409–429. Springer-Verlag, 1996.

[SAS98] Shamik Sharma, Anurag Acharya, and Joel Saltz. Deferred data-flow analysis. Technical Report TRCS98-38, University of California, Santa Barbara, December 1998.

[Tip94] Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1994.

A Proofs of Selected Theorems

A.1 Correctness of SVS

Following two Lemmas are needed to prove Theorem 6, which is in turn needed to prove Theorem 7.

Lemma 1 *Let* $\mathcal{C} = B^*(\mathcal{C}_0)$, $\mathcal{D} = atom(\mathcal{C})$, $\mathcal{I}_\mathcal{C} = gm(\mathcal{C})$, *and* $\mathcal{I}_\mathcal{D} = gm(\mathcal{D})$.

$$v \notin \mathcal{I}_\mathcal{D}(\mathcal{X}) \Leftrightarrow v \notin \mathcal{I}_\mathcal{D}(a) \textit{ for some } \mathcal{X} \subseteq a \textit{ in } \mathcal{C}$$

Proof. If $v \notin gm(\mathcal{C})(\mathcal{X})$ then $\mathcal{C}$ contains an upper bound on $\mathcal{X}$ of the form $\mathcal{X} \subseteq se$ such that $v \notin gm(\mathcal{C})(se)$. Thus, $v \notin \mathcal{I}_\mathcal{D}(\mathcal{X}) \Leftrightarrow v \notin \mathcal{I}_\mathcal{D}(a), \mathcal{X} \subseteq a$ in $\mathcal{D}$. This implies that $v \notin \mathcal{I}_\mathcal{D}(\mathcal{X}) \Leftrightarrow v \notin \mathcal{I}_\mathcal{D}(a)$ for some $\mathcal{X} \subseteq a$ in $\mathcal{C}$. $\square$

Lemma 2 *Let* $\mathcal{C} = B^*(\mathcal{C}_0)$, $\mathcal{D} = atom(\mathcal{C})$, $\mathcal{I}_\mathcal{C} = gm(\mathcal{C})$, *and* $\mathcal{I}_\mathcal{D} = gm(\mathcal{D})$.

$$\mathcal{I}_\mathcal{D} \subseteq \mathcal{I}_\mathcal{C}$$

Proof. We prove that $\mathcal{I}_\mathcal{D}$ is a model of $\mathcal{C}$, i.e., $v \notin \mathcal{I}_\mathcal{D}(se) \Rightarrow v \notin \mathcal{I}_\mathcal{D}(\mathcal{X})$.

- $\mathcal{X} \subseteq ae$ in $\mathcal{C}$

  $\Rightarrow$ trivially true because $\mathcal{X} \subseteq ae$ in $\mathcal{D}$

- $\mathcal{X} \subseteq \mathcal{Y}$ in $\mathcal{C}$ and $v \notin \mathcal{I}_{\mathcal{D}}(\mathcal{Y})$

  $\Rightarrow$ $v \notin \mathcal{I}_{\mathcal{D}}(ae)$ for some $\mathcal{Y} \subseteq ae$ in $\mathcal{C}$ (by Lemma 1)
  $\Rightarrow$ $\mathcal{X} \subseteq ae$ in $\mathcal{C}$ because $\mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq a$ in $\mathcal{C}$
  $\Rightarrow$ $\mathcal{X} \subseteq ae$ in $\mathcal{D}$
  $\Rightarrow$ $v \notin \mathcal{I}_{\mathcal{D}}(\mathcal{X})$

□

Theorem 6 *Let $\wp$ be a term with one free variable α, e be a sub-expression of $\wp$, V be a set of values, and $\wp \rhd_i \mathcal{C}_i$. Then $[\![atom(B^*(\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}))]\!]$ is the greatest model of $\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$.*

Proof. Let $\mathcal{C}' = \mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$. We prove $gm(\mathcal{C}') = gm(B^*(\mathcal{C}'))$ by showing that B always adds constraints that preserve the $gm(\mathcal{C}')$.

$$[\text{TRANS}]\ \frac{\mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq ae}{\mathcal{X} \subseteq ae}$$

$\mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq ae$ mean $\mathcal{I}(\mathcal{X}) \subseteq \mathcal{I}(ae)$, and so does $\mathcal{X} \subseteq ae$

$$[\text{TRANS-CON}_1]\ \frac{\kappa\, \mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \kappa\, \mathcal{Z}}{\mathcal{X} \subseteq \mathcal{Z}}$$

$\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ means $\mathcal{I}(\mathcal{X}) \subseteq \{v \mid \kappa\, v \in \mathcal{I}(\mathcal{Y})\}$, and $\mathcal{Y} \subseteq \kappa\, \mathcal{Z}$ means $\mathcal{I}(\mathcal{Y}) \subseteq \{\kappa\, v \mid v \in \mathcal{I}(\mathcal{Z})\}$.
Thus, $\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq \kappa\, \mathcal{Z}$ mean $\mathcal{I}(\mathcal{X}) \subseteq \mathcal{I}(\mathcal{Z})$, and so does $\mathcal{X} \subseteq \mathcal{Z}$.

$$[\text{TRANS-CON}_2]\ \frac{\kappa\, \mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \kappa\, \top}{\mathcal{X} \subseteq \top}$$

$\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ means $\mathcal{I}(\mathcal{X}) \subseteq \{v \mid \kappa\, v \in \mathcal{I}(\mathcal{Y})\}$, and $\mathcal{Y} \subseteq \kappa\, \top$ means $\mathcal{I}(\mathcal{Y}) \subseteq \{\kappa\, v \mid v \in \mathit{Val}\}$.
Thus, $\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq \kappa\, \top$ mean $\mathcal{I}(\mathcal{X}) \subseteq \mathit{Val}$, and so does $\mathcal{X} \subseteq \top$.

$$[\text{TRANS-CON}_3]\ \frac{\kappa\, \mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \overline{\kappa'}\, \top}{\mathcal{X} \subseteq \top}$$

$\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ means $\mathcal{I}(\mathcal{X}) \subseteq \{v \mid \kappa\, v \in \mathcal{I}(\mathcal{Y})\}$, and $\mathcal{Y} \subseteq \overline{\kappa'}\, \top$ means $\mathcal{I}(\mathcal{Y}) \subseteq \{\kappa''\, v \mid \kappa'' \neq \kappa', v \in \mathit{Val}\}$.
Since $\kappa \neq \kappa'$, $\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq \overline{\kappa'}\, \top$ mean $\mathcal{I}(\mathcal{X}) \subseteq \mathit{Val}$, and so does $\mathcal{X} \subseteq \top$.

$$[\text{TRANS-CON}_4]\ \frac{\kappa\, \mathcal{X} \subseteq \mathcal{Y} \quad \mathcal{Y} \subseteq \top}{\mathcal{X} \subseteq \top}$$

$\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ means $\mathcal{I}(\mathcal{X}) \subseteq \{v \mid \kappa\, v \in \mathcal{I}(\mathcal{Y})\}$, and $\mathcal{Y} \subseteq \top$ means $\mathcal{I}(\mathcal{Y}) \subseteq \{v \mid v \in \mathit{Val}\}$.
Since $\{v \mid \kappa\, v \in \mathit{Val}\} = \mathit{Val}$, $\kappa\, \mathcal{X} \subseteq \mathcal{Y}$ and $\mathcal{Y} \subseteq \top$ mean $\mathcal{I}(\mathcal{X}) \subseteq \mathit{Val}$, and so does $\mathcal{X} \subseteq \top$.

Now, we must prove $gm(B^*(\mathcal{C}_i)) = gm(atom(B^*(\mathcal{C}_i)))$. $gm(B^*(\mathcal{C}_i)) \subseteq gm(atom(B^*(\mathcal{C}_i)))$ because $atom(B^*(\mathcal{C}_i)) \subseteq B^*(\mathcal{C}_i)$, and $gm(atom(B^*(\mathcal{C}_i))) \subseteq gm(B^*(\mathcal{C}_i))$ by Lemma 2. □

Following Lemma is needed in proving Theorem 7.

Lemma 3 *Let $\wp \rhd_i \mathcal{C}_i$ and $\mathcal{I}$ be a greatest model of $\mathcal{C}_i \cup \{\mathcal{X}_0 \subseteq V_0\}$. For all $e \in \wp$, $\mathcal{I} \vdash e \leadsto v$ implies $v \in \mathcal{I}(\mathcal{X}_e)$ and furthermore, $\mathcal{I}$ is a safe set environment with respect to $[\nu/\alpha]\wp$, where $\nu \in \mathcal{I}(\alpha)$.*

Proof. By induction on proof tree size of set-based semantics of $\mathcal{I} \vdash e \leadsto v$

1. Base cases

   [VAR] $e = x$

   $\mathcal{I}(x) \vdash x \leadsto v \Rightarrow v \in \mathcal{I}(\mathcal{X}_x)$ (by semantics)

   [LAM] $e = \lambda x.e'$

   $\mathcal{I} \vdash \lambda x.e' \leadsto \lambda x.e' \Rightarrow \lambda x.e' \in \mathcal{I}(\mathcal{X}_e)$ (by Def of $\mathcal{I}$ and $\rhd_i$)

2. Induction Steps

   [CON] $e = \texttt{con}\ \kappa\ e$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathcal{I}(\mathcal{X}_e) = V} & \\
\Rightarrow & \mathcal{I}(\mathcal{X}_1) = \{v \mid \kappa\, v \in V\} & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} \\
\Rightarrow & (\mathcal{I} \vdash e_1 \leadsto v' \Rightarrow v' \in \{v \mid \kappa\, v \in V\}) & \text{(by I.H.)} \\
\Rightarrow & (\mathcal{I} \vdash \texttt{con}\ \kappa\ e_1 \leadsto v \Rightarrow v \in V) & \text{(by semantics)}
\end{array}
$$

   [DECON] $e = \texttt{decon}\ \kappa\ e_1$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathcal{I}(\mathcal{X}_e) = V} & \\
\Rightarrow & \mathcal{I}(\mathcal{X}_1) = \{\kappa\, v \mid v \in V\} & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} \\
\Rightarrow & (\mathcal{I} \vdash e_1 \leadsto v' \Rightarrow v' \in \{\kappa\, v \mid v \in V\}) & \text{(by I.H.)} \\
\Rightarrow & (\mathcal{I} \vdash \texttt{decon}\ \kappa\ e_1 \leadsto v \Rightarrow v \in V) & \text{(by semantics)}
\end{array}
$$

   [APP] $e = e_1\ e_2$

$$
\begin{array}{llll}
\multicolumn{2}{l}{\mathcal{I}(\mathcal{X}_e) = V} & & \\
\Rightarrow & \mathcal{I}(\mathcal{X}_{e'}) \subseteq V & & \\
 & \mathcal{I}(\mathcal{X}_1) \subseteq Lam_\wp(e_1) & & \\
 & \mathcal{I}(\mathcal{X}_2) \subseteq \mathcal{I}(\mathcal{X}_x) & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} & \\
\Rightarrow & (\mathcal{I} \vdash e_1 \leadsto \lambda x.e' \Rightarrow \lambda x.e' \in Lam_\wp(e_1)) & & \\
 & (\mathcal{I} \vdash e' \leadsto v \Rightarrow v \in V) & & \\
 & \mathcal{I}(x) \ni v', \text{if } \mathcal{I} \vdash e_2 \leadsto v' & \text{(by I.H.)} & \cdots (*) \\
 & & & \\
\Rightarrow & (\mathcal{I} \vdash e_1\ e_2 \leadsto v \Rightarrow v \in V) & \text{(by semantics)} &
\end{array}
$$

   [CASE] $e = \texttt{case}\ e_1\ \kappa\ e_2\ e_3$

$$
\begin{array}{llll}
\multicolumn{3}{l}{\mathcal{I}(\mathcal{X}_e) = V} & \\
1) & \Rightarrow & \mathcal{I}(\mathcal{X}_1) \subseteq \kappa\, \top & \\
 & & \mathcal{I}(\mathcal{X}_2) \subseteq V & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} \\
 & \Rightarrow & \mathcal{I} \vdash e_1 \leadsto v_1 \Rightarrow v_1 \in \kappa\, \top & \\
 & & \mathcal{I} \vdash e_2 \leadsto v_2 \Rightarrow v_2 \in V & \text{(by I.H.)} \\
 & \Rightarrow & \mathcal{I} \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \leadsto v \Rightarrow v \in V & \text{(by semantics)} \\
2) & \Rightarrow & \mathcal{I}(\mathcal{X}_1) \subseteq \overline{\kappa}\, \top & \\
 & & \mathcal{I}(\mathcal{X}_3) \subseteq V & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} \\
 & \Rightarrow & \mathcal{I} \vdash e_1 \leadsto v_1 \Rightarrow v_1 \in \overline{\kappa}\, \top & \\
 & & \mathcal{I} \vdash e_3 \leadsto v_3 \Rightarrow v_3 \in V & \text{(by I.H.)} \\
 & \Rightarrow & \mathcal{I} \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \leadsto v \Rightarrow v \in V & \text{(by semantics)} \\
3) & \Rightarrow & \mathcal{I}(\mathcal{X}_2) \subseteq V & \\
 & & \mathcal{I}(\mathcal{X}_3) \subseteq V & \text{(by Def. of } \mathcal{I} \text{ and } \rhd_i\text{)} \\
 & \Rightarrow & \mathcal{I} \vdash e_2 \leadsto v_2 \Rightarrow v_2 \in V & \\
 & & \mathcal{I} \vdash e_3 \leadsto v_3 \Rightarrow v_3 \in V & \text{(by I.H.)} \\
 & \Rightarrow & \mathcal{I} \vdash \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \leadsto v \Rightarrow v \in V & \text{(by semantics)}
\end{array}
$$

A set environment is safe with respect to a closed expression e_0, if it contains every binding which may occur in execution of e_0 [Hei93]. $(*)$ shows that $\mathcal{I}$ is safe with repsect to $[\nu/\alpha]\wp$ because for every application $e_1\ e_2$, $\mathcal{I}(\mathcal{X}_x) \supseteq \mathcal{I}(\mathcal{X}_2)$ where $Lam_\wp(e_1) \ni \lambda x.e'$ and $\mathcal{I}(\alpha) \ni \nu$. $\square$

Theorem 7 (Correctness of SVS) *Let $\wp$ be a term with one free variable α, and e be a sub-expression of $\wp$. If $\Sigma \in SVS_\wp(e \subseteq V)$ then $\forall \nu \in \Sigma(\alpha) : ([\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_e] \Rightarrow v \in V)$.*

Proof. Σ is a greatest model of $\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$ by Theorem 6. Thus by Lemma 3, $\Sigma \vdash e \rightsquigarrow v \Rightarrow v \in \Sigma(e)$ and Σ is a safe set environment with respect to $[\nu/\alpha]\wp$. $\Sigma(e) \subseteq V$ because Σ is a greatest model of $\mathcal{C}_i \cup \{\mathcal{X}_e \subseteq V\}$.

Therefore, $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_e]$

$$
\begin{array}{lll}
\Rightarrow & \Sigma \vdash e \rightsquigarrow v & \text{(because } \Sigma \text{ is safe w.r.t. } [\nu/\alpha]\wp) \\
\Rightarrow & v \in \Sigma(e) & \text{(by Lemma 3)} \\
\Rightarrow & v \in V & \text{(because } \Sigma \text{ is a greatest model)}
\end{array}
$$

□

A.2 Safety of *psba*

Theorem 8 (Safety of *psba*) *If $[\nu/\alpha]\wp \stackrel{*}{\longmapsto} \mathcal{E}[v_l]$ then $v \in psba(\wp)(l)$.*

Proof. We prove the safety of *psba* by showing that *psba* is a safe approximation of sba_ν. We define two continous functions $\mathcal{F}_\nu$ and $\mathcal{P}_\nu$ that correspond [CC95] to the closure operations $sba_\nu(\wp)$ and $psba_\nu(\wp)$, respectively. Then, we prove by the fixpoint induction that the least fixpoint of $\mathcal{P}_\nu$ is larger than or equal to the least fixpoint of $\mathcal{F}_\nu$.

The program $\wp$'s constraints $\mathcal{C}$ is constructed as $\triangleright \wp : \mathcal{C}$. Then $sba_\nu(\wp)$ is equal to the least fixpoint *lfp* $\mathcal{F}_\nu$ of continuous function $\mathcal{F}_\nu \colon (Vars(\mathcal{C}) \to 2^{Val}) \to (Vars(\mathcal{C}) \to 2^{Val})$ [CC95]

$$
\begin{array}{lll}
\mathcal{F}_\nu(\rho)(\mathcal{X}_l) & = & \text{case } e \text{ of} \\
\lambda x.e' & : & \{\lambda x.e'\} \\
x & : & \{v \mid e_1\ e_2 \in \wp, \lambda x.e' \in \rho(\mathcal{X}_1), v \in \rho(\mathcal{X}_2)\} \\
\alpha & : & \{\nu\} \\
e_1\ e_2 & : & \{v \mid \lambda x.e_3 \in \rho(\mathcal{X}_1), v \in \rho(\mathcal{X}_3)\} \\
\texttt{con}\ \kappa\ e_1 & : & \{\kappa(v) \mid v \in \rho(\mathcal{X}_1)\} \\
\texttt{decon}\ \kappa\ e_1 & : & \{v \mid \kappa(v) \in \rho(\mathcal{X}_1)\} \\
\texttt{case}\ e_1\ \kappa\ e_2\ e_3 & : & \{v \mid v \in \rho(\mathcal{X}_2), \kappa(v') \in \rho(\mathcal{X}_1)\} \\
 & & \cup\{v \mid v \in \rho(\mathcal{X}_3), \kappa'(v') \in \rho(\mathcal{X}_1), \kappa' \neq \kappa\}
\end{array}
$$

We define another continous function $\mathcal{P}_\nu \colon (Vars(\mathcal{C}) \to 2^{Val}) \to (Vars(\mathcal{C}) \to 2^{Val})$ such that $psba_\nu(\wp)$ is the least fixpoint *lfp* $\mathcal{P}_\nu$ of $\mathcal{P}_\nu$. Expression $\overline{e}$ indicates that its set-constraint is input-independent, and $\underline{e}$ indicates that its set-constraint is input-dependent: $\mathcal{X}_e \supseteq se_\square$. Expressions without the marks are those that we don't have to differentiate. Note that, except for the e_1 in $\texttt{case}\ e_1\ \kappa\ e_2\ e_3$, $\mathcal{P}_\nu(\varphi)(\overline{e})$ is exactly same as $\mathcal{P}_\nu(\varphi)(\underline{e})$.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{P}_\nu(\varphi)(\mathcal{X}_e) = \text{case } e \text{ of}} \\
\texttt{1} & : & \{1\} \\
\lambda x.e' & : & \{\lambda x.e'\} \\
\alpha & : & \{\nu\} \\
e_1\ e_2 & : & \{v \mid \lambda x.e' \in \varphi(\mathcal{X}_1), v \in \varphi(\mathcal{X}'_e)\} \\
\texttt{con}\ \kappa\ e_1 & : & \{\kappa(v) \mid v \in \varphi(\mathcal{X}_1)\} \\
\texttt{decon}\ \kappa\ e_1 & : & \{v \mid \kappa(v) \in \varphi(\mathcal{X}_1)\} \\
\texttt{case}\ \overline{e_1}\ \kappa\ e_2\ e_3 & : & \{v \mid v \in \varphi(\mathcal{X}_2), \kappa(v') \in \varphi(\mathcal{X}_1)\} \\
 & & \cup\{v \mid v \in \varphi(\mathcal{X}_3), \kappa'(v') \in \varphi(\mathcal{X}_1)\} \\
\texttt{case}\ \underline{e_1}\ \kappa\ e_2\ e_3 & : & \{v \mid v \in \varphi(\mathcal{X}_2), \nu \notin \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \overline{\kappa}\ \top)(\alpha)\} \\
 & & \cup\{v \mid v \in \varphi(\mathcal{X}_3), \nu \notin \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \kappa\ \top)(\alpha)\}\ ,
\end{array}
$$

where

$$
\begin{array}{ll}
\underline{e}, & \text{if } \mathcal{X}_e \supseteq se_\square \text{ after compile time preparation} \\
\overline{e}, & \text{otherwise}
\end{array}
$$

We prove $Q(\mathit{lfp}\ \mathcal{P}_\nu, \mathit{lfp}\ \mathcal{F}_\nu)$ by the fixpoint induction, where the assertion $Q(\varphi, \rho)$ is:

$$(\forall e \in \wp . \varphi(\mathcal{X}_e) \supseteq \rho(\mathcal{X}_e)) \wedge (\varphi \subseteq \mathit{lfp}\ \mathcal{P}_\nu) \wedge (\rho \subseteq \mathit{lfp}\ \mathcal{F}_\nu).$$

Base case $Q(\emptyset, \emptyset)$ is trivially true.

We prove that $Q(\mathcal{P}_\nu(\varphi), \mathcal{F}_\nu(\rho))$ holds given the induction hypothesis $Q(\varphi, \rho)$. That is, we need to show $\mathcal{P}_\nu(\varphi)(\mathcal{X}_e) \supseteq \mathcal{F}_\nu(\rho)(\mathcal{X}_e)$.

[CON] $e = \texttt{con}\ \kappa\ e_1$.

$$\begin{array}{lcll} \mathcal{P}_\nu(\varphi)(\mathcal{X}_e) & = & \{\kappa\, v \mid v \in \varphi(\mathcal{X}_1)\} & \text{(by definition)} \\ & \supseteq & \{\kappa\, v \mid v \in \rho(\mathcal{X}_1)\} & \text{(by I.H.)} \\ & = & \mathcal{F}_\nu(\rho)(\mathcal{X}_e) & \text{(by definition)} \end{array}$$

Other cases are similarly proved except for the [CASE].

[CASE] $e = \texttt{case}\ \underline{e_1}\ \kappa\ e_2\ e_3$.

$$\begin{array}{lll} \mathcal{P}_\nu(\varphi)(\mathcal{X}_e) & & \\ \quad = & \{v \mid v \in \varphi(\mathcal{X}_2), \nu \notin \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \overline{\kappa}\ \top)(\alpha)\} & \\ & \cup\{v \mid v \in \varphi(\mathcal{X}_3), \nu \notin \mathrm{SVS}_\wp(\mathcal{X}_1 \subseteq \kappa\ \top)(\alpha)\} & \text{(by definition)} \\ \quad \supseteq & \{v \mid v \in \varphi(\mathcal{X}_2), \kappa\, v' \in (\mathit{lfp}\ \mathcal{F}_\nu)(\mathcal{X}_1)\} & \\ & \cup\{v \mid v \in \varphi(\mathcal{X}_3), \kappa'\ v' \in (\mathit{lfp}\ \mathcal{F}_\nu)(\mathcal{X}_1), \kappa' \neq \kappa\} & (*) \\ \quad \supseteq & \{v \mid v \in \rho(\mathcal{X}_2), \kappa\, v' \in \rho(\mathcal{X}_1)\} & \\ & \cup\{v \mid v \in \rho(\mathcal{X}_3), \kappa'\ v' \in \rho(\mathcal{X}_1), \kappa' \neq \kappa\} & \text{(by I.H.)} \\ \quad = & \mathcal{F}_\nu(\rho)(\mathcal{X}_e). & \text{(by definition)} \end{array}$$

$(*)$ is true because, by Theorem 4 and Theorem 7, $v \in \mathit{sba}_\nu(\mathcal{X}) \wedge v \notin V \Rightarrow \nu \notin \mathrm{SVS}_\wp(\mathcal{X} \subseteq V)(\alpha)$. $\square$