Parameterized Partial Evaluation

CHARLES CONSEL and SIAU CHENG KHOO
Yale University

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs.

General Terms: Languages, Theory

Additional Key Words and Phrases: On-line and off-line partial evaluation, program specialization w.r.t. symbolic values

1. INTRODUCTION

Besides specializing programs with respect to concrete values, it is often necessary to specialize programs with respect to abstract values, i.e., static properties such as signs, ranges, and types. Specializing programs with respect to static properties is a natural extension of partial evaluation and significantly contributes towards adapting partial evaluation to larger varieties of applications. This idea was first investigated by Haraldsson [14] and carried out in practice with a system called Redfun in the late seventies. This system partially evaluates Interlisp programs. It manipulates symbolic values such as data types to describe the possible values of a variable and a processed expression.

Although the work on Redfun certainly started in the right direction, it has some limitations: (1) the static properties cannot be defined by the user; they are fixed; (2) the approach is not formally defined: no safety condition for the definition of symbolic values, no finiteness criteria for fixpoint iteration, etc.; and (3) because Redfun is an *on-line* partial evaluator—the treatment of the

This is an expanded version of a paper that appeared in the *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation.* The research was supported in part by NSF and DARPA grants CCR-8809919 and N00014-88-K-0573, respectively. The second author was also supported by a National University of Singapore Overseas Graduate Scholarship.
Authors' addresses: C. Consel, Pacific Software Research Center, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 19600 N.W. Von Neumann Drive, Beaverton, OR 97006-1999; S. C. Khoo, Dept. of Information Systems and Computer Science, National University of Singapore, 10 Kent Ridge Crescent, Singapore 0511, Republic of Singapore.

program is determined as it gets processed—and consists of numerous symbolic values and program transformations, it is computationally expensive. As a by-product, Redfun could not be self-applied as noticed in [10, 14], and thus, the partial-evaluation process could not be improved.

This paper introduces *parameterized partial evaluation*, a generic form of partial evaluation parameterized with respect to user-defined static properties. We develop an algebraic framework to enable modular definition of static properties. More specifically, from a concrete algebra, an abstract algebra called a *facet* is defined; it is composed of an abstract domain—capturing the properties of interest—and a set of abstract primitives that operate on this domain.

The safety criteria of this abstraction are captured by the notion of *facet mapping*. This mapping is defined using abstract interpretation [2, 15]. It relates two algebras with a suitable abstraction function. However, unlike abstract interpretation, not only does a facet define primitive functions that compute static properties, but it also defines ones that use abstract values to trigger computations at partial-evaluation time.

Furthermore, it is possible to capture the partial-evaluation behavior of primitive functions as a facet; this is achieved by considering an algebra whose domain is syntactic terms and operations are primitive functions.

In "conventional" partial evaluation [4], efficiency is achieved by an *off-line* strategy that consists of splitting the partial-evaluation process into two phases: *binding-time analysis* that statically determines the static and dynamic expressions of a program given a known/unknown division of its inputs; and *specialization* which processes a program driven by the binding-time information and the concrete values. Thus, the binding-time information of a program can be used for specialization as long as the input values match the known/unknown pattern given for binding-time analysis. Besides improving the specialization phase, an *off-line* partial evaluator enables realistic self-application [16].

Our framework is general enough to capture off-line partial evaluation. Just as a binding-time analysis is used to compute static/dynamic properties, we introduce a *facet analysis* to statically compute properties. A specializer can then use the result of facet analysis in the same way that it used the result of binding-time analysis previously to trigger computations. Because the facet analysis is performed statically, the specialization phase is kept simple, as before, in contrast with an on-line strategy that performs everything at once.

Our approach overcomes the limitations (1), (2) and (3) mentioned above. Let us summarize the new contributions of this paper.

—*Facet mapping* provides a uniform abstraction methodology for relating domains and primitive operations at three levels of evaluation: standard evaluation, on-line and off-line partial evaluation.

—The notion of *facet* offers a formal framework for introducing user-defined static properties: a facet is a safe abstraction of a concrete algebra.

—Partial evaluation can now be *parameterized* with respect to any number of facets, each facet encapsulating properties of interest for any given application.

—*Facet analysis*, another novel aspect, allows facet computation to be lifted from partial evaluation keeping the specialization phase simple (unlike conventional program transformation systems). Indeed, not only does the facet analysis statically determine which properties trigger computations, but it also selects the corresponding reduction operations prior to specialization. This makes it possible to achieve self-application and improve the specialization process.

1.1 Overview

The paper is organized as follows. Section 2 briefly introduces conventional partial evaluation. Section 3 describes the abstraction methodology used to define properties of interest. Section 4 presents on-line parameterized partial evaluation. In particular, Section 4.1 presents the notion of facet together with examples and Section 4.4 describes the semantics of on-line parameterized partial evaluation. Section 5 presents off-line parameterized partial evaluation. We introduce the notion of abstract facet in Section 5.1 and present facet analysis in Section 5.4. Section 6 presents an example of on-line and off-line parameterized partial evaluation. Section 7 describes the related work. Finally, in Section 8, this work is put into perspective. The proofs are given in Appendix A.

1.2 Notation

Most of our notation is that of standard denotational semantics. A domain $\mathbf{D}$ is a *pointed cpo*—a chain-complete partial order with a least element $\perp_D$ (called "bottom"). As is customary, during a computation $\perp_D$ means "not yet calculated" [15]. A domain has a binary ordering relation denoted by $\sqsubseteq_D$. The infix least upper bound (lub) operator for the domain $\mathbf{D}$ is written $\sqcup_D$; its prefix form, which computes the lub of a set of elements, is denoted $\bigsqcup_D$. Thus we have that for all $d \in \mathbf{D}$, $\perp_D \sqsubseteq_D d$ and $\perp_D \sqcup_D d = d$. Domain subscripts are often omitted, as in $\perp \sqcup d$, when they are clear from context.

A domain $\mathbf{D}$ is a *lattice* if for all $x, y \in \mathbf{D}$, $x \sqcup y$ and $x \sqcap y$ exists, where $\sqcap$ is the infix greatest lower bound (glb) operator for $\mathbf{D}$. Any lattice $\mathbf{D}$ has a maximum element $\top$ (called "top") such that for all $d \in \mathbf{D}$, $d \sqsubseteq_D \top_D$ and $\top_D \sqcap d = d$. A lattice $\mathbf{D}$ is *complete* if $\bigsqcup X$ and $\bigsqcap X$ exist for every subset $X \subseteq \mathbf{D}$. A domain is *flat* if all its elements apart from $\perp$ are incomparable with each other. Analogously, a lattice is *flat* if all its elements apart from $\perp$ and $\top$ are incomparable with each other.

The notation "$d \in \mathbf{D} = \cdots$" defines the domain (or set) $\mathbf{D}$ with "typical element" d, where $\cdots$ provides the domain specification usually via some combination of the following domain constructions: $\mathbf{D}_\perp$ denotes the domain $\mathbf{D}$ lifted with a new least element $\perp$. $\mathbf{D}_1 \to \mathbf{D}_2$ denotes the domain of all *continuous functions* from $\mathbf{D}_1$ to $\mathbf{D}_2$. $\mathbf{D}_1 + \mathbf{D}_2$ and $\mathbf{D}_1 \times \mathbf{D}_2$ denotes the separated sum and product, respectively, of the domains $\mathbf{D}_1$ and $\mathbf{D}_2$. $\mathbf{D}_1 \otimes \mathbf{D}_2$ denotes the *smashed product* of the domains $\mathbf{D}_1$ and $\mathbf{D}_2$; its elements are defined by the function, *smashed*, such that

$$\begin{array}{lcll} smashed & : & \mathbf{D}_1 \times \mathbf{D}_2 \rightarrow \mathbf{D}_1 \otimes \mathbf{D}_2 & \\ smashed(d,e) & = & \langle d_1, d_2 \rangle & \text{if } (d_1 \neq \bot_{D_1}) \text{ and } (d_2 \neq \bot_{D_2}) \\ & & \bot_{D_1 \otimes D_2} & \text{otherwise.} \end{array}$$

All domain/subdomain coercions are omitted when clear from context.

The ordering on functions $f, f' \in \mathbf{D}_1 \rightarrow \mathbf{D}_2$ is defined in the standard way: $f \sqsubseteq f' \Leftrightarrow (\forall d \in \mathbf{D}_1) f(d) \sqsubseteq f'(d)$. A function $f \in \mathbf{D}_1 \rightarrow \mathbf{D}_2$ is *monotonic* iff it satisfies $(\forall d, d' \in \mathbf{D}_1)\ d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$; it is *continuous* if in addition it satisfies $f(\sqcup\{d_i\}) = \sqcup\{f(d_i)\}$ for any chain $\{d_i\} \subseteq \mathbf{D}_1$. A function $f \in \mathbf{D}_1 \rightarrow \mathbf{D}_2$ is said to be *strict* if $f(\bot_{D_1}) = \bot_{D_2}$. An element $d \in \mathbf{D}$ is a *fixpoint* of $f \in \mathbf{D} \rightarrow \mathbf{D}$ iff $f(d) = d$; it is the *least fixpoint* if for every other fixpoint d', we have that $d \sqsubseteq d'$. The composition of function $f \in \mathbf{D}_1 \rightarrow \mathbf{D}_2$ with $f' \in \mathbf{D}_2 \rightarrow \mathbf{D}_3$ is denoted by $f' \circ f$.

Angle brackets are used for tupling. If $d = \langle d_1, \ldots, d_n \rangle \in \mathbf{D}_1 \times \cdots \times \mathbf{D}_n$, then for all $i \in \{1, \ldots, n\}$, $d \downarrow i$ denotes the ith element (that is, d_i) of d. For convenience, in the context of a smashed product, that is, $d \in \mathbf{D}_1 \otimes \cdots \otimes \mathbf{D}_n$, d^i denotes the ith element of d. Syntactic objects are consistently enclosed in double brackets, as in $[\![e]\!]$. Square brackets are used for environment update, as in $env[d/[\![x]\!]]$, which is equivalent to the function $\lambda v.$ if $v = [\![x]\!]$ *then d else env(v)*. The notation $env[d_i/[\![x_i]\!]]$ is shorthand for $env[d_1/[\![x_1]\!], \ldots, d_n/[\![x_n]\!]]$, where the subscript bounds are inferred from context. "New" environments are created by $\bot[d_i/[\![x_i]\!]]$. Similar notations are also used to denote cache, cache update and new cache respectively.

The paper describes three levels of evaluations: standard evaluation, on-line partial evaluation and off-line partial evaluation. A symbol s is noted $\hat{s}$ if it is used in on-line partial evaluation and $\tilde{s}$ in off-line partial evaluation. Symbols that refer to standard semantics are unannotated. Finally, for generality, any symbol used in either on-line or off-line partial evaluation is noted $\bar{s}$.

2. PRELIMINARIES

In this section we examine conventional partial evaluation for strict functional programs. For conciseness we only consider first order programs in this paper.

Let us first examine Figure 1 that displays the standard semantics for a first order functional language. As is customary, we will omit summand projections and injections. Domain **Values** is a sum of the basic semantic domains (we only consider the integer and boolean domains in this paper). Function $\mathscr{K}$ maps a constant to its semantic value; function $\mathscr{K}_P$ defines the usual semantic operations for primitive operators. Domain **FunEnv** maps function names to their meaning. The meaning of a program is the meaning of f_1. We assume all functions (and primitive operations) have the same arity.

Figure 2 defines the semantics of a simple partial evaluator for programs written in our language. It is based on existing approaches ([4, 22, 7], for example). The figure only highlights aspects of the semantics relevant to later

1. Syntactic Domains

$$\begin{array}{lcll}
c & \in & \textbf{Const} & \text{Constants} \\
x & \in & \textbf{Var} & \text{Variables} \\
p & \in & \textbf{Po} & \text{Primitive Operators} \\
f & \in & \textbf{Fn} & \text{Function Names} \\
e & \in & \textbf{Exp} & \text{Expressions}
\end{array}$$

$$\begin{array}{rcl}
e & ::= & c \mid x \mid p\ (e_1, \cdots, e_n) \mid f\ (e_1, \cdots, e_n) \mid \mathit{if}\ e_1\ e_2\ e_3 \\
\text{Prog} & ::= & \{f_i(x_1, \cdots, x_n) = e_i\} \quad (\text{f}_1 \text{ is the main function})
\end{array}$$

2. Semantic Domains

$$\begin{array}{lcl}
b & \in & \textbf{Values} = \textbf{Int} + \textbf{Bool} \\
\rho & \in & \textbf{Env} = \textbf{Var} \rightarrow \textbf{Values} \\
\Theta & \in & \textbf{FunEnv} = \textbf{Fn} \rightarrow \textbf{Values}^n \rightarrow \textbf{Values}
\end{array}$$

3. Valuation Functions

$$\begin{array}{rcl}
\mathcal{E}_{Prog} & : & \textbf{Prog} \rightarrow \textbf{Values} \\
\mathcal{E} & : & \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{FunEnv} \rightarrow \textbf{Values} \\
\mathcal{K} & : & \textbf{Const} \rightarrow \textbf{Values} \\
\mathcal{K}_P & : & \textbf{Po} \rightarrow \textbf{Values}^n \rightarrow \textbf{Values}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E}_{Prog}\ [\![\{\ f_i(x_1, \cdots, x_n) = e_i\}]\!] = & & \\
\quad \Theta\ [\![f_1]\!]\ \mathit{whererec}\ \Theta & = & \bot[(\lambda(b_1, \cdots, b_n)\ .\ \mathcal{E}\ [\![e_i]\!]\ (\bot[b_k/x_k])\ \Theta)/f_i] \\
\mathcal{E}\ [\![c]\!]\ \rho\ \Theta & = & \mathcal{K}\ [\![c]\!] \\
\mathcal{E}\ [\![x]\!]\ \rho\ \Theta & = & \rho\ [\![x]\!] \\
\mathcal{E}\ [\![p(e_1, \cdots, e_n)]\!]\ \rho\ \Theta & = & \mathcal{K}_P\ [\![p]\!]\ (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta, \cdots,\ \mathcal{E}\ [\![e_n]\!]\ \rho\ \Theta) \\
\mathcal{E}\ [\![\mathit{if}\ e_1\ e_2\ e_3]\!]\ \rho\ \Theta & = & (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta) \rightarrow\ \mathcal{E}\ [\![e_2]\!]\ \rho\ \Theta,\ \mathcal{E}\ [\![e_3]\!]\ \rho\ \Theta \\
\mathcal{E}\ [\![f(e_1, \cdots, e_n)]\!]\ \rho\ \Theta & = & \Theta\ [\![f]\!]\ (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta, \cdots, \mathcal{E}\ [\![e_n]\!]\ \rho\ \Theta)
\end{array}$$

Fig. 1. Standard semantics of a first order language.

discussion. For example, we omit details about treatment of function calls[1] (unfolding and suspension). Because this treatment vastly differs from one partial evaluator to another, it is abstracted from the semantics by function *APP*.

Domain **Sf** defines a *cache* that keeps the specialization patterns of each function and maps these patterns to the corresponding specialized functions. Essentially, this achieves *instantiation* and *folding* as in [5], and ensures uniqueness of specialized functions (i.e., there exists exactly one specialized function in the cache for each suspended function call.) To keep track of each specialization, partial evaluation is single-threaded with respect to the cache, as is done customarily. This causes the evaluation order of the language to be explicit. Function *MkProg* constructs a residual program from the specialized functions contained in the cache.

[1] Note that unfolding a function call consists of replacing the call by the result of partially evaluating the function body, in an environment binding the parameters to the value of the arguments. Suspending of a function call consists of replacing it by a run-time call to a variant of the called function. This variant is produced by specializing the function with respect to the value of the static arguments.

1. Syntactic Domains
   (defined in Figure 1)
2. Semantics Domains

$$\begin{array}{lcll}
\rho & \in & \mathbf{Env} & = \mathbf{Var} \rightarrow \mathbf{Exp} \\
\varpi & \in & \mathbf{FnEnv} & = \mathbf{Fn} \rightarrow \mathbf{Exp}^n \rightarrow (\mathbf{Exp} \times \mathbf{Sf}) \\
\sigma & \in & \mathbf{Sf} & = (\mathbf{Fn} \times \mathbf{Const}^*) \rightarrow \mathbf{Exp}
\end{array}$$

3. Valuation Functions

$$\begin{array}{rcl}
\mathcal{SPE}_{Prog} & : & \mathbf{Prog} \rightarrow \mathbf{Input} \rightarrow \mathbf{Prog}_\perp \\
\mathcal{SPE} & : & \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{FnEnv} \rightarrow \mathbf{Sf} \rightarrow (\mathbf{Exp} \times \mathbf{Sf}) \\
\mathcal{SK}_P & : & \mathbf{Po} \rightarrow \mathbf{Exp}^n \rightarrow \mathbf{Exp} \\
MkProg & : & \mathbf{Sf} \rightarrow \mathbf{Prog} \quad \text{(omitted)}
\end{array}$$

$$\begin{array}{l}
\mathcal{SPE}_{Prog}\,[\![\{\ f_i(x_1,\cdots,x_n) = e_i\}]\!]\,\langle i_1,\cdots,i_n\rangle = \\
\quad MkProg\ (\ \mathcal{SPE}\,[\![f_1(x_1,\cdots,x_n)]\!]\,(\perp[i_k/[\![x_k]\!]])\ \varpi\ \perp)\downarrow 2 \\
\quad wherer ec\ \ \varpi = \perp[(\lambda(\phi_1,\cdots,\phi_n,\sigma)\ .\ \mathcal{SPE}\,[\![e_i]\!]\,(\perp[\phi_k/[\![x_k]\!]])\ \varpi\ \sigma)/[\![f_i]\!]]
\end{array}$$

$$\begin{array}{lcl}
\mathcal{SPE}\,[\![c]\!]\,\rho\,\varpi\,\sigma & = & \langle [\![c]\!],\ \sigma\rangle \\
\mathcal{SPE}\,[\![x]\!]\,\rho\,\varpi\,\sigma & = & \langle \rho\,[\![x]\!],\ \sigma\rangle \\
\mathcal{SPE}\,[\![p(e_1,\cdots,e_n)]\!]\,\rho\,\varpi\,\sigma & = & \langle \mathcal{SK}_P\,[\![p]\!]\,(e'_1,\cdots,e'_n),\ \sigma_n\rangle \\
 & & \quad where\quad \langle e'_1,\ \sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma \\
 & & \qquad\qquad\quad \vdots \qquad\qquad\quad \vdots \\
 & & \qquad\qquad \langle e'_n,\ \sigma_n\rangle = \mathcal{SPE}\,[\![e_n]\!]\,\rho\,\varpi\,\sigma_{n-1} \\
\mathcal{SPE}\,[\![if\ e_1\ e_2\ e_3]\!]\,\rho\,\varpi\,\sigma & = & (e'_1 \in \mathbf{Const}) \rightarrow \\
 & & \quad (\mathcal{K}\,e'_1) \rightarrow \mathcal{SPE}\,[\![e_2]\!]\,\rho\,\varpi\,\sigma_1,\ \mathcal{SPE}\,[\![e_3]\!]\,\rho\,\varpi\,\sigma_1, \\
 & & \quad \langle [\![if\ e'_1\ e'_2\ e'_3]\!],\ \sigma_3\rangle \\
 & & \quad where\quad \langle e'_2,\ \sigma_2\rangle = \mathcal{SPE}\,[\![e_2]\!]\,\rho\,\varpi\,\sigma_1 \\
 & & \qquad\qquad\ \ \langle e'_3,\ \sigma_3\rangle = \mathcal{SPE}\,[\![e_3]\!]\,\rho\,\varpi\,\sigma_2 \\
 & & where\ \langle e'_1,\ \sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma \\
\mathcal{SPE}\,[\![f(e_1,\cdots,e_n)]\!]\,\rho\,\varpi\,\sigma & = & APP\,[\![f]\!]\,e'_1\cdots e'_n\,\sigma_n\,\varpi \\
 & & \quad where\quad \langle e'_1,\ \sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma \\
 & & \qquad\qquad\quad \vdots \qquad\qquad\quad \vdots \\
 & & \qquad\qquad \langle e'_n,\ \sigma_n\rangle = \mathcal{SPE}\,[\![e_n]\!]\,\rho\,\varpi\,\sigma_{n-1}
\end{array}$$

$$\mathcal{SK}_P\,[\![p]\!](e_1,\cdots,e_n) = \bigwedge_{i=1}^{n}(e_i \in \mathbf{Const}) \rightarrow \mathcal{K}^{-1}\,(\mathcal{K}_p[\![p]\!]\,((\mathcal{K}\,e_1),\cdots,(\mathcal{K}\,e_n))),\ [\![p(e_1,\cdots,e_n)]\!]$$

Fig. 2. Simple partial evaluation semantics.

Because partial evaluation is a source-to-source program transformation, it operates on expressions. The set of expressions forms a flat domain, denoted by **Exp**.

Domain **FnEnv** recursively defines the meaning of each function. The monotonic function $\mathcal{K}^{-1}$ maps a value (e.g., integer and Boolean) back to its textual representation (**Exp**).

Partial evaluation subsumes standard evaluation. This is reflected, for instance, in the treatment of the primitive functions: when a primitive is called with constant arguments, its standard semantics is invoked. In general, an expression is completely evaluated when it solely depends on available data. Lastly, notice that in partial evaluation the primitive operators compute new values; in dealing with properties, we will want them to play a similar role.

With this preliminary material in hand, we are now ready to introduce parameterized partial evaluation.

3. THE ABSTRACTION METHODOLOGY

This section presents a general methodology to introduce abstract values in the partial-evaluation process. Sections 4 and 5 describe, respectively, how to instantiate this methodology for on-line and off-line partial evaluation, and provide examples for each instantiation.

In optimizing compilation, static properties are introduced to reason about a program prior to its execution. Computation of static properties is then defined by abstract versions of primitive functions. This structure (domain/operations) naturally prompted us to use an algebraic approach to model static properties. In fact, a concrete algebra corresponds to the notion of *semantic algebra* as defined in denotational semantics (e.g., [20]).

Definition 1 (*Semantic Algebra*). A semantic algebra, $[\mathbf{D}; \mathbf{O}]$, consists of a semantic domain $\mathbf{D}$, and a set of operations $\mathbf{O}$.

The operations of a semantic algebra are assumed to be monotonic.

Our approach consists of defining, from the semantic algebra, an abstract algebra composed of an abstract domain—capturing the properties of interest—and the set of abstract primitives operating on this domain. Using abstract interpretation [2, 15], this can be formally achieved by relating the two algebras with an abstraction function. Because we aim at addressing both on-line and off-line partial evaluation, a given algebra may be defined at three different levels—listed in increasing abstractness: standard semantics, on-line partial evaluation and off-line partial evaluation. These levels respectively define semantic algebras, facets, and abstract facets.

The rest of this section describes a general methodology to relate these different levels. In essence, this amounts to relating two algebras. To investigate this, we first discuss how to relate the domains and their operations in Sections 3.1 and 3.2, respectively. Then, this is formalized in Section 3.3 where the notion of relating two algebras is precisely defined together with safety criteria.

3.1 Relating Domains

Domains can be related using an *abstraction function* [9]. Such a function is strict and monotonic; it maps an initial domain into an abstract domain.

As a simple example, say we wish to introduce some symbolic computations on signs abstracted from the integer algebra $[\mathbf{D}; \mathbf{O}]$. To do so we first have to define an abstraction of the integer domain that captures the sign properties. A natural abstract domain is $\hat{\mathbf{D}} = \{\perp, pos, zero, neg, \top\}$. Domains $\hat{\mathbf{D}}$ and $\mathbf{D}$ are related by the following abstraction function.

$$
\begin{array}{lcll}
\hat{\alpha}_{\hat{D}} & : & \mathbf{D} \rightarrow \hat{\mathbf{D}} & \\
\hat{\alpha}_{\hat{D}}(x) & = & \perp_{\hat{D}} & \text{if } d = \perp_D \\
 & & \text{pos} & \text{if } d > 0 \\
 & & \text{zero} & \text{if } d = 0 \\
 & & \text{neg} & \text{if } d < 0
\end{array}
$$

This example is further developed in Section 4.1.

Technically, note that to facilitate the proof of Properties 4 and 7, abstraction functions are required to be $\perp$ *-reflecting* [1] (i.e., let f: $\mathbf{A} \to \mathbf{B}$, $fa = \perp_B \Rightarrow a = \perp_A$).

3.2 Relating Operations

In abstracting one algebra from another, not only do we want to relate a domain to an abstract domain but we also want to relate the operators to their abstract versions. More precisely, we want to formulate the safety condition of an approximation to an operator.

Essentially, relating two operators consists of relating their graphs. To this end, we distinguish two classes of operators. The first class is composed of operators *closed* under the carrier of the algebra. That is, for an algebra $[\mathbf{A}; \mathbf{O}]$, we say that $p \in \mathbf{O}$ is closed if and only if $p: \mathbf{A} \to \mathbf{A}$. Thus, the abstract version of a closed operator will be passed abstract value to compute new ones; this corresponds to an abstract primitive in abstract interpretation.

The second class of operators consists of those whose codomain is different from the carrier; they are referred to as *open*. Intuitively, abstract versions of open operators will use abstract values to perform actual computations. Interestingly, we can relate this division to optimizing compilation where, typically, a phase collects properties and another triggers optimizations using these properties.

For convenience, given an algebra $[\mathbf{A}; \mathbf{O}]$, $\mathbf{O}_o$ and $\mathbf{O}_c$ will denote the set of open and closed operators, respectively.

This division suggests that since an abstraction function relates the carriers of two algebras, it can also be used to relate an operator and its abstract version when this operator is closed under the carrier. However, this does not apply to open operators since their domain differs from their codomain. Since an operator may be defined at three different levels (standard semantics, on-line and off-line partial evaluation), its corresponding codomain will then have three different definitions: in the standard semantics, an operator belongs to a semantic algebra; both open and closed operators produce basic values (domain **Values**). In on-line partial evaluation, an operator belongs to a facet; when it is open it produces a constant provided it is called with appropriate values (see Section 4). In off-line partial evaluation, an operator belongs to an abstract facet; when the operator is open it mimics the facet operator and thereby produces a binding-time value (i.e., *Static* or *Dynamic*) (see Section 5).

Thus, in order to relate an open operator to its abstract version, we have to relate their codomains. To do so let us define the abstraction functions relating the three levels of definition of domain **Values**.

From standard semantics to on-line partial evaluation, we need to map basic values into their textual representation; this mapping is defined as follows.

$$\begin{array}{lcll} \hat{\tau} & : & \mathbf{Values} \to \widehat{\mathbf{Values}} & \\ \hat{\tau}(x) & = & \perp_{\widehat{Values}} & \text{if} \quad x = \perp_{Values} \\ & & \mathscr{K}^{-1} x & \text{otherwise.} \end{array}$$

Because **Values** is a sum of basic domains it is more convenient to define $\hat{\tau}$ as a family of abstraction functions indexed by the basic domain. That is, for each basic domain **D**, there is an abstraction function $\hat{\tau}_D$: $\mathbf{D} \rightarrow \widehat{\mathbf{Values}}$ defined. To keep the notation simple, we omit the indexing of function $\hat{\tau}$.

Domain $\widehat{\mathbf{Values}}$ consists of the set of constants denoted by **Const**, augmented with elements $\bot_{\widehat{Values}}$ and $\top_{\widehat{Values}}$; these two elements are respectively weaker and stronger than all the elements of **Const**. For convenience, we assume the functions defined on **Const** to be also defined on $\widehat{\mathbf{Values}}$ (e.g., function $\mathscr{K}$); this domain is further discussed in Section 4.

To investigate the relation between on-line partial evaluation and off-line partial evaluation, recall that conventional off-line partial evaluation consists of a binding-time analysis and a specializer. The binding-time domain, noted $\widetilde{\mathbf{Values}}$, is composed of the set {*Static*, *Dynamic*} lifted with a least element[2] $\bot_{\widetilde{Values}}$. This domain forms a chain, with ordering $\bot_{\widetilde{Values}} \sqsubseteq$ *Static* $\sqsubseteq$ *Dynamic*, and abstracts the on-line partial-evaluation process in the following way.

$$\begin{array}{lcll}
\tilde{\tau} & : & \widehat{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}} & \\
\tilde{\tau}(x) & = & \bot_{\widetilde{Values}} & \text{if} \quad x = \bot_{\widehat{Values}} \\
 & & \text{Static} & \text{if} \quad x \in \mathbf{Const} \\
 & & \text{Dynamic} & \text{otherwise.}
\end{array}$$

This reflects the fact that an expression is static if it partially evaluates to a constant.

3.3 Relating Algebras

Given this preliminary discussion, we can now formalize the notion of algebra abstraction.

Let $\alpha' = \{\alpha_{B'_i}: \mathbf{B}_i \rightarrow \mathbf{B}'_i\}$ be a family of abstraction functions, $[\mathbf{A}; \mathbf{O}]$ and $[\mathbf{A}'; \mathbf{O}']$ be two algebras and $\alpha_{A'}: \mathbf{A} \rightarrow \mathbf{A}'$ be an abstraction function. Then, $\alpha_{A'}: [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A}'; \mathbf{O}']$ is called a *facet mapping* with respect to α', and is defined as follows:

Definition 2 (*Facet Mapping*). $\alpha_{A'}: [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A}'; \mathbf{O}']$ is a facet mapping with respect to $\alpha' = \{\alpha_{B'_i}: \mathbf{B}_i \rightarrow \mathbf{B}'_i\}$ if and only if

(1) $\mathbf{A}'$ is a complete lattice of finite height;[3]
(2) $\forall p' \in \mathbf{O}'$, p' is monotonic;
(3) If $p \in \mathbf{O}$ is a closed operator, then p': $\mathbf{A}' \rightarrow \mathbf{A}'$ is its corresponding abstract version;

[2] Note that this three-point domain refines the usual two-point domain {*Static*, *Dynamic*} in that it allows us to detect functions in a program that are never invoked, and simple cases of nonterminating computations. Without the value $\bot_{\widetilde{Values}}$, these cases would be considered as *Static*.

[3] Notice that with a lattice of infinite height, a *widening* operator can be used to find fixpoints in a finite number of steps (see [9]).

(4) If $p \in \mathbf{O}$ is an open operator with functionality $\mathbf{A} \to \mathbf{B}_\iota$, where $\mathbf{B}_\iota$ is some domain different from $\mathbf{A}$, then p': $\mathbf{A}' \to \mathbf{B}'_\iota$ is its corresponding abstract version;

(5) $\forall p \in \mathbf{O}$ and its corresponding abstract version $p' \in \mathbf{O}'$
$\alpha_{A'} \circ p \sqsubseteq p' \circ \alpha_{A'}$ if p is a closed operator
$\alpha' \circ p \sqsubseteq p' \circ \alpha_{A'}$ if p is an open operator with functionality $\mathbf{A} \to \mathbf{B}_\iota$.

Notice that Condition 1 ensures termination in computing abstract values. Also, for simplicity, we only consider a limited form of heterogeneous algebra (Conditions 3 and 4): only the codomain of an operator can be different from the carrier of the algebra. Finally, Condition 5 defines the safety criteria of an approximation to an operator.

Given a facet mapping, we can succinctly describe the relationship between the components of two algebras by a *logical relation* [19, 15].

Definition 3 (*Logical Relation* $\sqsubseteq_{\alpha_{A'}}$). A facet mapping $\alpha_{A'}$: $[\mathbf{A};\mathbf{O}] \to [\mathbf{A}';\mathbf{O}']$ with respect to $\alpha' = \{\alpha_{B'_\iota}: \mathbf{B}_\iota \to \mathbf{B}'_\iota\}$ induces a logical relation $\sqsubseteq_{\alpha_{A'}}$ as follows.

(1) $\forall a \in \mathbf{A}, \forall a' \in \mathbf{A}'$: $a \sqsubseteq_{\alpha_{A'}} a' \Leftrightarrow \alpha_{A'}(a) \sqsubseteq_{A'} a'$;

(2) Let $p \in \mathbf{O}$ and $p' \in \mathbf{O}'$ be closed operators. Then, $p \sqsubseteq_{\alpha_{A'}} p' \Leftrightarrow \forall a \in \mathbf{A}$, $\forall a' \in \mathbf{A}'$: $a \sqsubseteq_{\alpha_{A'}} a' \Rightarrow p(a) \sqsubseteq_{\alpha_{A'}} p'(a')$;

(3) Let $p \in \mathbf{O}$ and $p' \in \mathbf{O}'$ be open operators and p: $\mathbf{A} \to \mathbf{B}_\iota$ for some domain $\mathbf{B}_\iota$. Then, $p \sqsubseteq_{\alpha_{A'}} p' \Leftrightarrow \forall a \in \mathbf{A}$, $\forall a' \in \mathbf{A}'$: $a \sqsubseteq_{\alpha_{A'}} a' \Rightarrow p(a) \sqsubseteq_{\alpha'} p'(a')$ where $\forall b \in \mathbf{B}, \forall b' \in \mathbf{B}'$: $b \sqsubseteq_{\alpha'} b' \Leftrightarrow \alpha_{B'}(b) \sqsubseteq_{B'} b'$.

Using this logical relation, we can reformulate the safety criteria expressed by Condition 5 of Definition 2 as follows.

PROPERTY 1. *Let* $\alpha_{A'}$: $[\mathbf{A};\mathbf{O}] \to [\mathbf{A}';\mathbf{O}']$ *be a facet mapping with respect to* $\alpha' = \{\alpha_{B'_\iota}: \mathbf{B}_\iota \to \mathbf{B}'_\iota\}$, $\forall p \in \mathbf{O}$ *and its corresponding abstract version* $p' \in \mathbf{O}'$, $p \sqsubseteq_{\alpha_{A'}} p'$.

Facet mapping provides a uniform abstraction methodology for introducing static properties, in the form of abstract algebras, at both the online and offline levels of partial evaluation. In the following two sections, we instantiate facet mapping to introduce static properties at these two levels. Each instantiation is illustrated by an example.

4. ON-LINE PARAMETERIZED PARTIAL EVALUATION

This section presents online parameterized partial evaluation. We first define the notion of facet by instantiating the abstraction methodology described in Section 3. We then describe online parameterized partial evaluation.

4.1 Facets

A facet captures symbolic computations performed in online partial evaluation. As a result, while a closed operator will compute new abstract values, an open operator will produce constants when provided with appropriate abstract values. Formally:

Definition 4 (*Facet*). A facet for a semantic algebra $[\mathbf{D};\mathbf{O}]$ is an algebra $[\hat{\mathbf{D}};\hat{\mathbf{O}}]$ defined by a facet mapping $\hat{\alpha}_{\hat{D}}$: $[\mathbf{D};\mathbf{O}] \to [\hat{\mathbf{D}};\hat{\mathbf{O}}]$ with respect to $\hat{\tau}$.

We refer to $\hat{\mathbf{D}}$ as the facet domain and $\hat{\mathbf{O}}$ as the set of facet operators. The use of facet mapping in the definition ensures the following property about the open operators of a facet.

PROPERTY 2. *For any open operator* $p \in \mathbf{O}$ *of arity* n, $\forall \hat{d}_i, \cdots, \hat{d}_n \in \hat{\mathbf{D}}$ *and* $\forall d_i \in \mathbf{D}$, *if* $d_i \sqsubseteq_{\hat{\alpha}_{\hat{D}}} \hat{d}_i$ $\forall i \in \{1, \cdots, n\}$, *then* $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \in \mathbf{Const} \wedge p(d_1, \cdots, d_n) \neq \perp \Rightarrow \hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \hat{\tau}(p(d_1, \cdots, d_n))$.

In essence, this property states that if an open operator of a facet yields a constant for some abstract values, this constant is the same as that produced by the concrete operator called with the corresponding concrete values. Notice that this equality only holds if the call to the concrete operator terminates. The concrete values d_i are those related to the abstract values $\hat{d}_i$ under the logical relation $\sqsubseteq_{\hat{\alpha}_{\hat{D}}}$.

However, for some values, an open operator of a facet may not yield a constant. Indeed, it may be passed abstract values too coarse to be of any use. This is illustrated in the example below.

As an example of a facet, say we wish to define a **Sign** facet from an integer algebra. The set of static properties would be $\{\perp, pos, zero, neg, \top\}$. Assume that the operators of this algebra are $\{+, <\}$. Then $+$ would be a closed operator: it operates on two sign values to compute a new one. However, $<$ is an open operator: it uses the abstract value of its arguments to trigger computation whenever possible (e.g., $\hat{<}(zero, pos) = [\![true]\!]$).

Example 1. Sign information forms a facet for semantic algebra $[\mathbf{D};\mathbf{O}] = [\mathbf{Int};\{+,<\}]$.

(1) $\hat{\mathbf{D}} = \{\perp, pos, zero, neg, \top\}$ with $\forall \hat{d} \in \hat{\mathbf{D}}. \perp \sqsubseteq \hat{d} \sqsubseteq \top$

(2) The abstraction function is

$$\begin{array}{lcll} \hat{\alpha}_{\hat{D}} & : & \mathbf{D} \to \hat{\mathbf{D}} & \\ \hat{\alpha}_{\hat{D}}(d) & = & \perp_{\hat{D}} & \text{if } d = \perp_D \\ & & \text{pos} & \text{if } d > 0 \\ & & \text{zero} & \text{if } d = 0 \\ & & \text{neg} & \text{if } d < 0 \end{array}$$

(3) $\hat{\mathbf{O}} = \hat{O}_o \cup \hat{O}_c$ where $\hat{O}_o = \{\hat{<}\}$ and $\hat{O}_c = \{\hat{+}\}$

(4) Facet operators

$$\begin{array}{l} \hat{+} : \hat{\mathbf{D}} \times \hat{\mathbf{D}} \to \hat{\mathbf{D}} \\ \hat{+} = \lambda(\hat{d}_1, \hat{d}_2).(\hat{d}_1 = \perp) \vee (\hat{d}_2 = \perp) \to \perp, \\ \qquad\qquad \hat{d}_1 = \text{zero} \to \hat{d}_2, \\ \qquad\qquad \hat{d}_2 = \text{zero} \to \hat{d}_1, \hat{d}_1 \sqcup \hat{d}_2 \end{array}$$

$$\hat{<} : \hat{\mathbf{D}} \times \hat{\mathbf{D}} \to \widehat{\mathbf{Values}}$$
$$\begin{aligned}\hat{<} = \lambda(\hat{d}_1, \hat{d}_2).(\hat{d}_1 &= \bot) \vee (\hat{d}_2 = \bot) \to \bot_{\widehat{Values}},\\
&(\hat{d}_1 = \text{pos}) \wedge (\hat{d}_2 \in \{\text{neg}, \text{zero}\}) \to [\![\mathit{false}]\!],\\
&(\hat{d}_1 = \text{zero}) \wedge (\hat{d}_2 = \text{pos}) \to [\![\mathit{true}]\!],\\
&(\hat{d}_1 = \text{zero}) \wedge (\hat{d}_2 \in \{\text{neg}, \text{zero}\}) \to [\![\mathit{false}]\!],\\
&(\hat{d}_1 = \text{neg}) \wedge (\hat{d}_2 \in \{\text{pos}, \text{zero}\}) \to [\![\mathit{true}]\!], \top_{\widehat{Values}}\end{aligned}$$

We can now explain further our approach and examine how the notion of facet achieves the parameterization of partial evaluation.

4.2 Product of Facets

Essentially, parameterized partial evaluation differs from the conventional partial evaluation in two aspects: it collects facet information and propagates the results of facet computations to all relevant facets. While the latter aspect is described explicitly in the new partial-evaluation model presented in Section 4.4, the former is captured by the notion of the *product of facets* defined in this section.

A product of facets captures the set of facets defined for a given semantic algebra. It consists of the product of facet domains and the set of facet operators. In particular, for each operator p, a *product operator*, noted ω_p, triggers each facet operator $\hat{p}_i$ with the corresponding abstract values. If p is a closed operator, the product operation yields a product of abstract values. Otherwise, it produces either a constant, $\bot_{\widehat{Values}}$ or $\top_{\widehat{Values}}$, depending on the abstract values available.

Definition 5 (*Product of Facets*). Let $\hat{\alpha}_i : [\mathbf{D}; \mathbf{O}] \to [\hat{\mathbf{D}}_i; \hat{\mathbf{O}}_i]$ for $i \in \{1, \ldots, m\}$ be the set of facet mappings defined for a semantic algebra $[\mathbf{D}; \mathbf{O}]$. Its product of facets, noted $[\hat{\mathscr{D}}; \hat{\Omega}]$, consists of two components:

(1) A domain $\hat{\mathscr{D}} = \hat{\mathbf{D}}_1 \otimes \cdots \otimes \hat{\mathbf{D}}_m \cong \prod_{i=1}^{m} \hat{\mathbf{D}}_i$; it is a smashed product of the facet domains;

(2) A set of product operators $\hat{\Omega}$ such that $\forall p \in \mathbf{O}$ and its corresponding product operator $\hat{\omega}_p \in \hat{\Omega}$,

(a) if $p \in \mathbf{O}$ is a closed operator, then
$p: \mathbf{D}^n \to \mathbf{D}$, and
$\hat{\omega}_p : \hat{\mathscr{D}}^n \to \hat{\mathscr{D}}$
$\hat{\omega}_p = \lambda(\hat{\delta}_1, \cdots, \hat{\delta}_n) \cdot \prod_{i=1}^{m} \hat{p}_i(\hat{\delta}_1^i, \cdots, \hat{\delta}_n^i)$;

(b) otherwise, $p \in \mathbf{O}$ is an open operator
$p: \mathbf{D}^n \to \mathbf{D}'$ for some domain $\mathbf{D}'$, and
$\hat{\omega}_p : \hat{\mathscr{D}}^n \to \widehat{\mathbf{Values}}$
$$\begin{aligned}\hat{\omega}_p = \lambda(\hat{\delta}_1, \cdots, \hat{\delta}_n) \cdot &(\exists j \in \{1, \cdots, m\} \text{ s.t. } \hat{d}_j = \bot_{\widehat{Values}}) \to \bot_{\widehat{Values}},\\
&(\exists j \in \{1, \cdots, m\} \text{ s.t. } \hat{d}_j \in \mathbf{Const}) \to \hat{d}_j, \top_{\widehat{Values}}\end{aligned}$$
where $\hat{d} = \langle \hat{p}_1(\hat{\delta}_1^1, \cdots, \hat{\delta}_n^1), \cdots, \hat{p}_m(\hat{\delta}_1^m, \cdots, \hat{\delta}_n^m) \rangle$.

Domain $\hat{\mathscr{D}}$ is partially ordered componentwise. The smashed product construction is used for this domain to ensure the notion of consistency explained below.

Although facets of a product are defined independently, the facet values, to which a program is specialized, must have some *consistency*. This notion of consistency can be motivated by the following example. Suppose that two facets are defined for the integer algebra: one facet describes the sign of an integer value (see Example 1), and the other indicates whether the value is odd or even. Then, a value such as $\langle zero, odd \rangle$ should not be considered as a valid facet value, since *zero* is an even number. Formally:

Definition 6. Let $[\hat{\mathscr{D}}; \hat{\Omega}]$ be a product of facets of an algebra $[\mathbf{D}; \mathbf{O}]$; $\hat{\delta} \in \hat{\mathscr{D}}$ is consistent if and only if

$$\bigcap_{i=1}^{m} \left\{ d \in \mathbf{D} \mid d \sqsubseteq_{\hat{\alpha}_i} \hat{\delta}^i \right\} \text{ is neither the empty set nor } \{\bot\}.$$

Each set of concrete values corresponds to a particular facet property; it is defined by the logical relation $\sqsubseteq_{\hat{\alpha}_i}$. Notice that by definition of the relation $\sqsubseteq_{\hat{\alpha}_i}$, the above intersection will at least yield the singleton $\{\bot\}$; thus such singleton set must not imply consistency. In essence, the above definition ensures that a product of abstract values represents an actual subdomain of $\mathbf{D}$.

Technically, note that the smashed product construction is used to conveniently eliminate inconsistent values such as $\langle \bot, odd \rangle$.

We assume that a program is always specialized with respect to consistent products of facet values. By definition of a facet, the consistency property is preserved by the open and closed operators. This property contributes to the correctness of the following lemma, which states that if there is more than one facet that produce concrete values, those values are equal.

LEMMA 3. *Let* $[\hat{\mathscr{D}}; \hat{\Omega}]$ *be a product of facets and* $p \in \mathbf{O}$ *be an open operator. If* $\exists j, k \in \{1, \cdots, m\}$ $(j \neq k)$ *and* $\hat{\delta}_1, \cdots, \hat{\delta}_n \in \hat{\mathscr{D}}$ *such that both* $\hat{p}_j(\hat{\delta}_1^j, \cdots, \hat{\delta}_n^j) \in \mathbf{Const}$ *and* $\hat{p}_k(\hat{\delta}_1^k, \cdots, \hat{\delta}_n^k) \in \mathbf{Const}$, *then* $\hat{p}_j(\hat{\delta}_1^j, \cdots, \hat{\delta}_n^j) = \hat{p}_k(\hat{\delta}_1^k, \cdots, \hat{\delta}_n^k)$.

Last, we show, below, a property about the product operators.

PROPERTY 4. *All operators defined in the product of facets,* $[\hat{\mathscr{D}}; \hat{\Omega}]$, *are monotonic.*

We have seen how properties of interest can be formally introduced via a facet and described how facets could be combined to form a product of facets. Let us now explore the generality of the approach. In particular, we want to examine how partial evaluation of primitive operations can itself be captured by a facet.

4.3 Partial-Evaluation Facet

So far, we have used the notion of facet to introduce symbolic computations drawn from a semantic algebra defined in the standard semantics. In fact, the same notion can also be used to define a facet that captures the traditional partial-evaluation behavior of primitives. It is called the *partial-evaluation facet*. More specifically, for a given semantic algebra, the corresponding partial-evaluation facet will define its standard semantics whenever it is passed constant arguments. The partial-evaluation facet is defined as follows.

Definition 7 (*Partial-Evaluation Facet*). The partial-evaluation facet of a semantic algebra $[\mathbf{D};\mathbf{O}]$ is defined by the facet mapping $\hat{\alpha}_{\widehat{Values}}: [\mathbf{D};\mathbf{O}] \to [\widehat{\mathbf{Values}};\hat{\mathbf{O}}]$.

(1) $\hat{\alpha}_{\widehat{Values}}: \mathbf{D} \to \widehat{\mathbf{Values}}$
$\hat{\alpha}_{\widehat{Values}} \equiv \hat{\tau}_D$

(2) $\forall \hat{p} \in \hat{\mathbf{O}}$ of arity n
$\hat{p}: \widehat{\mathbf{Values}}^n \to \widehat{\mathbf{Values}}$
$$\hat{p} = \lambda(\hat{d}_1,\cdots,\hat{d}_n).\exists i \in \{1,\cdots,n\} \text{ s.t. } \hat{d}_i = \bot_{\widehat{Values}} \to \bot_{\widehat{Values}},$$
$$\wedge_{i=1}^{n}(\hat{d}_i \in \mathbf{Const}) \to \hat{\tau}(\mathscr{K}_p[\![p]\!](d_1,\cdots,d_n)),\ \top_{\widehat{Values}}$$
where $d_i = (\mathscr{K}\hat{d}_i)\ i \in \{1,\cdots,n\}$.

In fact, the abstraction function $\hat{\alpha}_{\widehat{Values}}$ corresponds to the abstraction function $\hat{\tau}_D$ defined for domain $\mathbf{D}$ and taken from the family of abstraction functions $\hat{\tau}$ given in Section 3.2. This function maps a value into its textual representation (that is, a constant).

PROPERTY 5. *The partial-evaluation facet* (*Definition* 7) *is a facet.*

Notice that, just as any other facet operator, a partial-evaluation facet operator produces value $\top_{\widehat{Values}}$ when it is passed to values that are too coarse (that is, nonconstant values).

We can now define the semantics of parameterized partial evaluation.

4.4 Semantics of On-line Parameterized Partial Evaluation

Since this semantics aims at defining partial evaluation, we assume that the partial-evaluation facet always exists. Thus, because a partial-evaluation facet is defined for each semantic domain, it will be assigned to the first component of every product of facets. A sum of these products of facets is noted $\widehat{\mathscr{S}\mathscr{D}}$; each summand correpsonds to a semantic algebra.

Figure 3 displays the parameterized partial-evaluation semantics. For simplicity, we assume that every product of facets contains m facets (including the partial-evaluation facet). Also, we assume that user-supplied facets are globally defined, that is, the corresponding abstraction functions and product operators are globally defined.

The skeleton of this semantics is the same as the traditional partial-evaluation semantics displayed in Figure 2. This is not surprising, since introducing facets essentially enriches the semantic algebras of this semantics.

For a product of facets $\hat{\mathscr{D}}$, $\hat{\alpha}_{\hat{D}_i}$ denotes the ith abstraction function. Besides computing facet values, the partial evaluator has to construct the residual

1. Semantic Domains

$$\begin{array}{lcl}
\widehat{\delta} & \in & \widehat{\mathcal{SD}} = \sum_{j=1}^{s} \widehat{\mathcal{D}}_j \quad \textit{where } \widehat{\mathcal{D}}_j = (\widehat{\mathbf{D}}_{j1} \otimes \cdots \otimes \widehat{\mathbf{D}}_{jm}) \textit{ and } s \textit{ is the number of basic domains} \\
e' & \in & \mathbf{Exp} \\
\rho & \in & \mathbf{Env} = \mathbf{Var} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}}) \\
\varpi & \in & \mathbf{FnEnv} = \mathbf{Fn} \rightarrow (\mathbf{Exp}^n \times \widehat{\mathcal{SD}}^n \times \mathbf{Sf}) \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf}) \\
\sigma & \in & \mathbf{Sf} = (\mathbf{Fn} \times \mathbf{Exp}^n \times \widehat{\mathcal{SD}}^n) \rightarrow \mathbf{Exp}
\end{array}$$

2. Valuation Functions

$$\begin{array}{lcl}
\mathcal{PE}_{Prog} & : & \mathbf{Prog} \rightarrow \mathbf{Exp}^n \rightarrow \widehat{\mathcal{SD}}^n \rightarrow \mathbf{Prog}_\perp \\
\mathcal{PE} & : & \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{FnEnv} \rightarrow \mathbf{Sf} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf}) \\
\widehat{\mathcal{K}}_P & : & \mathbf{Po} \rightarrow \mathbf{Exp}^n \rightarrow \widehat{\mathcal{SD}}^n \rightarrow \mathbf{Sf} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf})
\end{array}$$

$$\begin{array}{l}
\mathcal{PE}_{Prog}\, \llbracket \{ f_i(x_1, \cdots, x_n) = e_i \} \rrbracket\, \langle e'_1, \cdots, e'_n \rangle\, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle = \\
\quad (MkProg\ \sigma)\ \textit{whererec}\ \langle -, \sigma \rangle = \mathcal{PE}\, \llbracket f_1(x_1, \cdots, x_n) \rrbracket\, (\perp[\langle e'_k, \widehat{\delta}_k \rangle / \llbracket x_k \rrbracket])\ \varpi\ \perp \\
\qquad\qquad\qquad\qquad\quad \varpi = \perp[(\lambda(\langle e'_1, \cdots, e'_n \rangle, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle, \sigma)\ .\ \mathcal{PE}\, \llbracket e_i \rrbracket\, [\langle e', \widehat{\delta}_k \rangle / \llbracket x_k \rrbracket]\ \varpi\ \sigma) / \llbracket f_i \rrbracket
\end{array}$$

$$\mathcal{PE}\, \llbracket c \rrbracket\, \rho\, \varpi\, \sigma = \widehat{\mathcal{K}}\, \llbracket c \rrbracket\, \sigma$$

$$\mathcal{PE}\, \llbracket x \rrbracket\, \rho\, \varpi\, \sigma = \langle e', \widehat{\delta}, \sigma \rangle \quad \textit{where } \langle e', \widehat{\delta} \rangle = \rho\, \llbracket x \rrbracket$$

$$\begin{array}{lcl}
\mathcal{PE}\, \llbracket p(e_1, \cdots, e_n) \rrbracket\, \rho\, \varpi\, \sigma & = & \widehat{\mathcal{K}}_P\, \llbracket p \rrbracket\, \langle e'_1, \cdots, e'_n \rangle\, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle\, \sigma_n \\
& & \quad \textit{where} \quad \langle e'_1, \widehat{\delta}_1, \sigma_1 \rangle = \mathcal{PE}\, \llbracket e_1 \rrbracket\, \rho\, \varpi\, \sigma \\
& & \qquad\qquad\quad \vdots \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\quad \langle e'_n, \widehat{\delta}_n, \sigma_n \rangle = \mathcal{PE}\, \llbracket e_n \rrbracket\, \rho\, \varpi\, \sigma_{n-1}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{PE}\, \llbracket if\ e_1\ e_2\ e_3 \rrbracket\, \rho\, \varpi\, \sigma & = & (e'_1 \in \mathbf{Const}) \rightarrow (\mathcal{K}\, e'_1) \rightarrow \mathcal{PE}\, \llbracket e_2 \rrbracket\, \rho\, \varpi\, \sigma_1,\ \mathcal{PE}\, \llbracket e_3 \rrbracket\, \rho\, \varpi\, \sigma_1, \\
& & \qquad \langle \llbracket if\ e'_1\ e'_2\ e'_3 \rrbracket, \widehat{\delta}, \sigma_3 \rangle \\
& & \qquad\quad \textit{where}\ \langle e'_2, \widehat{\delta}_2, \sigma_2 \rangle = \mathcal{PE}\, \llbracket e_2 \rrbracket\, \rho\, \varpi\, \sigma_1 \\
& & \qquad\qquad\qquad \langle e'_3, \widehat{\delta}_3, \sigma_3 \rangle = \mathcal{PE}\, \llbracket e_3 \rrbracket\, \rho\, \varpi\, \sigma_2 \\
& & \qquad\qquad\qquad \widehat{\delta} = \widehat{\delta}_2 \sqcup \widehat{\delta}_3 \\
& & \textit{where}\ \langle e'_1, \widehat{\delta}_1, \sigma_1 \rangle = \mathcal{PE}\, \llbracket e_1 \rrbracket\, \rho\, \varpi\, \sigma
\end{array}$$

$$\begin{array}{lcl}
\mathcal{PE}\, \llbracket f(e_1, \cdots, e_n) \rrbracket\, \rho\, \varpi\, \sigma & = & APP\, \llbracket f \rrbracket\, \langle e'_1, \cdots, e'_n \rangle\, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle\, \sigma_n\, \varpi \\
& & \quad \textit{where} \quad \langle e'_1, \widehat{\delta}_1, \sigma_1 \rangle = \mathcal{PE}\, \llbracket e_1 \rrbracket\, \rho\, \varpi\, \sigma \\
& & \qquad\qquad\quad \vdots \qquad\qquad\qquad \vdots \\
& & \qquad\qquad\quad \langle e'_n, \widehat{\delta}_n, \sigma_n \rangle = \mathcal{PE}\, \llbracket e_n \rrbracket\, \rho\, \varpi\, \sigma_{n-1}
\end{array}$$

$$\widehat{\mathcal{K}}\, \llbracket c \rrbracket\, \sigma = \langle \llbracket c \rrbracket, \langle \widehat{\alpha}_{\widehat{\mathcal{D}}_1}(d), \cdots, \widehat{\alpha}_{\widehat{\mathcal{D}}_m}(d) \rangle, \sigma \rangle \quad \textit{where } d = (\mathcal{K}\, c) \in \mathbf{D}$$

$$\begin{array}{lcl}
\widehat{\mathcal{K}}_P\, \llbracket p^c \rrbracket\, \langle e'_1, \cdots, e'_n \rangle\, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle\, \sigma & = & (\widehat{\delta} = \perp_{\widehat{\mathcal{D}}}) \rightarrow \langle \llbracket p^c(e'_1, \cdots, e'_n) \rrbracket, \perp_{\widehat{\mathcal{SD}}}, \sigma \rangle \\
& & (\widehat{\delta}^1 \in \mathbf{Const}) \rightarrow \langle \widehat{\delta}^1, \langle \widehat{\alpha}_{\widehat{\mathcal{D}}_1}(d), \cdots, \widehat{\alpha}_{\widehat{\mathcal{D}}_m}(d) \rangle, \sigma \rangle, \\
& & \qquad\qquad\qquad \langle \llbracket p^c(e'_1, \cdots, e'_n) \rrbracket, \widehat{\delta}, \sigma \rangle \\
& & \textit{where } p^c : \mathbf{D}^n \rightarrow \mathbf{D} \\
& & \qquad \widehat{\delta} = \widehat{\omega}_{p^c}(\widehat{\delta}_1, \cdots, \widehat{\delta}_n) \\
& & \qquad d = \mathcal{K}\, \widehat{\delta}^1
\end{array}$$

$$\begin{array}{lcl}
\widehat{\mathcal{K}}_P\, \llbracket p^o \rrbracket\, \langle e'_1, \cdots, e'_n \rangle\, \langle \widehat{\delta}_1, \cdots, \widehat{\delta}_n \rangle\, \sigma & = & (\hat{d} = \perp_{\widehat{Values}}) \rightarrow \langle e', \perp_{\widehat{\mathcal{SD}}}, \sigma \rangle, \\
& & \hat{d} \in \mathbf{Const} \rightarrow \langle \hat{d}, \langle \widehat{\alpha}_{\widehat{\mathcal{D}'_1}}(d), \cdots, \widehat{\alpha}_{\widehat{\mathcal{D}'_m}}(d) \rangle, \sigma \rangle \\
& & \qquad\qquad\qquad \langle e', \langle \top_{\widehat{\mathcal{D}'_1}}, \cdots, \top_{\widehat{\mathcal{D}'_m}} \rangle, \sigma \rangle \\
& & \textit{where } p^o : \mathbf{D}^n \rightarrow \mathbf{D}' \\
& & \qquad \hat{d} = \widehat{\omega}_{p^o}(\widehat{\delta}_1, \cdots, \widehat{\delta}_n) \\
& & \qquad d = \mathcal{K}\, \hat{d} \\
& & \qquad e' = \llbracket p^o(e'_1, \cdots, e'_n) \rrbracket
\end{array}$$

Fig. 3. On-line parameterized partial evaluation.

program and collect the specialized functions. This triple forms the codomain of the partial-evaluation function and is defined as $\mathbf{Exp} \times \widehat{\mathscr{S}\mathscr{D}} \times \mathbf{Sf}$. Closed and open operators are respectively noted p^c and p^o.

Notice that when an expression partially evaluates to a constant—because the expression is either a constant or a primitive called with appropriate values—functions $\hat{\mathscr{K}}$ and $\hat{\mathscr{K}}_P$ propagate this value to all facets in a product by invoking their corresponding abstraction function.

The following theorem asserts that any constant produced by partially evaluating a primitive call is always correct with respect to the standard semantics, modulo termination.

THEOREM 1. *Let* $[\hat{\mathscr{D}}; \hat{\Omega}]$ *be a product of facets (including the partial-evaluation facet) for an algebra* $[\mathbf{D}; \mathbf{O}]$. *Let* $c = (\mathscr{P}\mathscr{E}[\![p(x_1, \cdots, x_n)]\!] \bot [\langle [\![x_i]\!], \hat{\delta}_i \rangle / [\![x_i]\!]] \bot_{FnEnv} \bot_{Sf}) \downarrow 1$, *and* $v = (\mathscr{E}[\![p(x_1, \cdots, x_n)]\!] \bot [d_i / [\![x_i]\!]] \bot_{FnEnv})$ *where* $d_i \in \bigcap_{j=1}^{m} \{d \in \mathbf{D} \mid d \sqsubseteq_{\hat{\alpha}_j} \hat{\delta}_i^j\}$, *for* $i \in \{1, \ldots, n\}$. *Then,*

$$(c \in \mathbf{Const}) \text{ and } v \neq \bot \Rightarrow c = \hat{\tau}(v).$$

Finally, let us point out that on-line partial evaluation as defined in Figure 3 provides a less complete treatment of conditional expressions than the one described in Redfun [14]. Indeed, Redfun is able to extract properties from the predicate of a conditional expression. Then, these properties and their negation are propagated to the consequent and alternative branches, respectively. This is somewhat similar to constraints in logic programming. We are currently investigating this issue to possibly incorporate the notion of constraints in our approach.

5. OFF-LINE PARAMETERIZED PARTIAL EVALUATION

As discussed earlier, in an on-line strategy all decisions about how to process an expression are made at partial-evaluation time. This makes it possible to determine precise treatment based, for example, on concrete values. However, this is computationally expensive because the partial evaluator must analyze the context of the computation—the available data—to select the appropriate program transformation. This operation is repeatedly performed when processing recursive functions.

In conventional partial evaluation efficiency is achieved by an *off-line* strategy which splits the partial-evaluation phase into binding-time analysis and specialization. In particular, the binding-time analysis only computes the static/dynamic property. In off-line parameterized partial evaluation, we generalize the binding-time analysis to *facet analysis*: a phase that statically computes facet information. Consequently, the task of program specialization reduces to following the information yielded by the facet analysis.

To present off-line parameterized partial evaluation, we follow the approach used in defining on-line parameterized partial evaluation: we introduce the concept of abstract facet in Section 5.1, describe the product of abstract facets in Section 5.2, define the binding-time facet in Section 5.3, and last, describe facet analysis in Section 5.4.

5.1 Abstract Facets

To lift facet computation from partial evaluation, we need to define a suitable abstraction of this process. In particular, we need to define an abstraction of a facet that enables facet computation to be performed prior to specialization. The resulting facet is called an *abstract facet* and is defined in this section.

Not surprisingly, an abstract facet has the same structure as a facet. In particular it has two classes of operators: open and closed. Similar to a facet, a closed operator of an abstract facet is passed abstract values and computes new ones. As for an open operator, it mimics the corresponding facet operator: it uses abstract values to produce binding-time values. More precisely, instead of a constant it produces the binding-time value *Static*, and instead of $\top_{\widehat{Values}}$ it produces *Dynamic*.

Just as a facet is defined from a semantic algebra, an abstract facet is defined from a facet. Formally:

Definition 8 (*Abstract Facet*). An abstract facet $[\tilde{\mathbf{D}}; \tilde{\mathbf{O}}]$ of a facet $[\hat{\mathbf{D}}; \hat{\mathbf{O}}]$ is defined by a facet mapping $\tilde{\alpha}_{\tilde{\mathrm{D}}}\colon [\hat{\mathbf{D}}; \hat{\mathbf{O}}] \to [\tilde{\mathbf{D}}; \tilde{\mathbf{O}}]$ with respect to $\tilde{\tau}$.

This definition leads to the following property about open operators.

PROPERTY 6. *For any open operator* $\hat{p} \in \hat{\mathbf{O}}$ *of arity* n, $\forall \tilde{d}_i, \cdots, \tilde{d}_n \in \tilde{\mathbf{D}}$ *and* $\forall \hat{d}_i \in \hat{\mathbf{D}}$, *if* $\hat{d}_i \sqsubseteq_{\tilde{\alpha}_{\tilde{D}}} \tilde{d}_i$ *for* $i \in \{1, \cdots, n\}$, *then* $(\tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n) = \mathit{Static}) \Rightarrow \hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \sqsubseteq_{\widehat{Values}} c$ *with* $c \in$ **Const**.

This property states that, when an open operator of an abstract facet maps some properties into the value *Static*, the open operator of the corresponding facet will yield a constant value at specialization time, modulo termination.

As an example of an abstract facet, say we wish to define a **Sign** abstract facet from the **Sign** facet (Example 1). This will amount to determining, prior to specialization, whether sign computation can produce constants. If so, the specialization phase will collect sign information and trigger the open operators that produced the value *Static* at facet analysis time.

Example 2. The abstract facet for the **Sign** facet $[\hat{\mathbf{D}}; \hat{\mathbf{O}}]$ is defined as follows.

(1) $\tilde{\mathbf{D}} = \hat{\mathbf{D}}$ (similar to Example 1)

(2) $\tilde{\alpha}_{\tilde{D}}$ is simply the identity mapping between $\hat{\mathbf{D}}$ and $\tilde{\mathbf{D}}$.

(3) $\tilde{O} = \{\tilde{<}, \tilde{+}\}$ where $\tilde{+}$ has the same functionality as $\hat{+}$ and $\tilde{<}$ is defined as follows.

$$\begin{aligned}
\tilde{<} \quad &: \quad \tilde{\mathbf{D}} \times \tilde{\mathbf{D}} \to \widetilde{\mathbf{Values}} \\
\tilde{<} \quad &= \quad \lambda(a, b).\; a = \bot \vee b = \bot \to \bot_{\widetilde{Values}}, \\
&\qquad a = \text{pos} \wedge (b \in \{\text{neg}, \text{zero}\}) \to \text{Static}, \\
&\qquad a = \text{zero} \wedge b = \text{pos} \to \text{Static}, \\
&\qquad a = \mathit{zero} \wedge (b \in [\text{neg}, \text{zero}\}) \to \text{Static}, \\
&\qquad a = \text{neg} \wedge (b \in \{\text{pos}, \text{zero}\}) \to \text{Static}, \text{Dynamic}
\end{aligned}$$

5.2 Product of Abstract Facets

As in on-line parameterized partial evaluation, we now define the *product of abstract facets*. It captures the set of abstract facets derived from the set of facets defined for a given semantic algebra.

Definition 9 (*Product of Abstract Facets*). Let $\tilde{\alpha}_i$: $[\hat{\mathbf{D}}_i; \hat{\mathbf{O}}_i] \to [\tilde{\mathbf{D}}_i; \tilde{\mathbf{O}}_i]$ for $i \in \{1, \ldots, m\}$ be the set of facet mappings defined for the facets of a semantic algebra $[\mathbf{D}; \mathbf{O}]$. Its product of abstract facets, noted $[\tilde{\mathscr{D}}, \tilde{\Omega}]$, consists of two components:

(1) A domain $\tilde{\mathscr{D}} = \prod_{i=1}^{m} \tilde{\mathbf{D}}_i$ is a smashed product of the abstract-facet domains;

(2) A set of product operators $\tilde{\Omega}$ such that $\forall p \in \mathbf{O}$ and its corresponding product operator $\tilde{\omega}_p \in \tilde{\Omega}$;

   (a) if $\tilde{p}$ is a closed operator, then
   $p\colon \mathbf{D}^n \to \mathbf{D}$, and
   $\tilde{\omega}_p\colon \tilde{\mathscr{D}}^n \to \tilde{\mathscr{D}}$
   $\tilde{\omega}_p = \lambda(\tilde{\delta}_1, \cdots, \tilde{\delta}_n) \cdot \prod_{i=1}^{m} \tilde{p}_i(\tilde{\delta}_1^i, \cdots, \tilde{\delta}_n^i)$.

   (b) otherwise, $\tilde{p} \in \tilde{\mathbf{O}}$ is an open operator, and
   $p\colon \mathbf{D}^n \to \mathbf{D}'$ for some domain $\mathbf{D}'$, and
   $\tilde{\omega}_p\colon \tilde{\mathscr{D}}^n \to \widetilde{\mathbf{Values}}$
   $\tilde{\omega}_p = \lambda(\tilde{\delta}_1, \cdots, \tilde{\delta}_n) \cdot (\exists j \in \{1, \cdots, m\} \text{ s.t. } \tilde{d}_j = \bot_{\widetilde{Values}}) \to \bot_{\widetilde{Values}},$
   $(\exists j \in \{1, \cdots, m\} \text{ s.t. } \tilde{d}_j = \text{Static}) \to \text{Static}, \text{Dynamic}$
   where $\tilde{d} = \langle \tilde{p}_1(\tilde{\delta}_1^1, \cdots, \tilde{\delta}_n^1), \cdots, \tilde{p}_m(\tilde{\delta}_1^m, \cdots, \tilde{\delta}_n^m) \rangle$.

The domain $\tilde{\mathscr{D}}$ is partially ordered componentwise. Since all the product components are of finite height by definition, the product domain is also of finite height.

PROPERTY 7. *All operators defined in the product of abstract facets,* $[\tilde{\mathscr{D}}; \tilde{\Omega}]$, *are monotonic.*

5.3 Binding-Time Facet

While the partial-evaluation semantics of algebraic operators is captured by a facet, the computation of their binding-time values can similarly be captured by the notion of abstract facet. Such an abstract facet is called a *binding-time facet*.

Definition 10 (*Binding-Time Facet*). The binding-time facet of a partial-evaluation facet $[\widetilde{\mathbf{Values}}; \hat{\mathbf{O}}]$ is defined by the facet mapping $\tilde{\alpha}_{\widetilde{Values}}$: $[\widetilde{\mathbf{Values}}; \hat{\mathbf{O}}] \to [\widetilde{\mathbf{Values}}; \tilde{\mathbf{O}}]$

(1) $\tilde{\alpha}_{\widetilde{Values}}\colon \widetilde{\mathbf{Values}} \to \widetilde{\mathbf{Values}}$
   $\tilde{\alpha}_{\widetilde{Values}} \equiv \tilde{\tau}$

(2) $\forall \tilde{o} \in \tilde{\mathbf{O}}$ of arity n
   $\tilde{o}\colon \widetilde{\mathbf{Values}}^n \to \widetilde{\mathbf{Values}}$
   $\tilde{o} = \lambda(\tilde{d}_1, \cdots, \tilde{d}_n) \cdot \exists j \in \{1, \ldots, n\} \text{ s.t. } \tilde{d}_j = \bot_{\widetilde{Values}} \to \bot_{\widetilde{Values}},$
   $\bigwedge_{i=1}^{n} (\tilde{d}_i = \text{Static}) \to \text{Static}, \text{Dynamic}$

PROPERTY 8. *The binding-time facet (Definition 10) is an abstract facet.*

Not surprisingly, Definition 10 captures the primitive functions of a conventional binding-time analysis. As a result, not only does the facet analysis compute user-defined abstract values but it also computes binding-time values, just as a binding-time analysis.

5.4 Facet Analysis

We are now ready to examine the facet analysis. It is essentially a conventional binding-time analysis, as described in [22] for example, extended to compute facet information. Analogous to the definition of parameterized online partial evaluation, we assume the binding-time facet to be always defined. The main semantic domain used by the analysis is denoted by $\widetilde{\mathscr{SD}}$, which is a sum of products of abstract facets—each summand corresponds to a semantic algebra. The binding-time facet is assigned to the first component of each product.

Facet analysis is displayed in Figure 4. It generalizes the traditional binding-time analysis (e.g., [22, 18]) to compute abstract-facet information. This analysis aims at collecting abstract-facet information for each function in a given program; this forms the *facet signature* of a function. More precisely, a facet signature of a function consists of a product of abstract-facet values for the arguments and its corresponding result; it is defined as $\widetilde{\mathscr{SD}}^{n+1}$. The result of the analysis (domain **SigEnv**) is a function mapping each user-defined function in the program to its facet signature.

The valuation function $\tilde{\mathscr{E}}$ maps each user-defined function into its abstract version. The resulting abstract functions are then used by the valuation function $\tilde{\mathscr{A}}$ to compute the facet signatures. As usual, computation is accomplished via fixpoint iteration. Functions $\tilde{\mathscr{K}}$ and $\tilde{\mathscr{K}}_P$ perform the abstract computation on constants and primitive operators. This is similar to functions $\hat{\mathscr{K}}$ and $\hat{\mathscr{K}}_P$ defined in Figure 3. Finally, note that fixpoint iteration is performed over the domains $\widetilde{\mathscr{SD}}$ and **SigEnv**. Since these domains are of finite height and operations over these domains are monotonic, a fixpoint will be reached in a finite number of steps.

6. AN EXAMPLE

This section illustrates further parameterized partial evaluation with an example of a program computing the inner product of two vectors. After describing this program, we examine its online and offline partial evaluation when the size of the vectors is known. In this example we consider vectors of floating point numbers.

One can think of a vector as an abstract data type **V** consisting of a set of operators **O** listed below.

MktVec: **Int** $\rightarrow$ **V**
MktVec creates an empty vector of the specified size
UpdVec: **V** $\times$ **Int** $\times$ **Float** $\rightarrow$ **V**
UpdVec updates an element

1. Syntactic Domains
   (defined in Figure 1)
2. Semantic Domains

$$\begin{array}{llll}
\widetilde{\delta} & \in & \widetilde{\mathcal{SD}} & = \sum_{j=1}^{s} \widetilde{\mathcal{D}}_j \;\; \textit{where } \widetilde{\mathcal{D}}_j = (\widetilde{\mathbf{D}}_{j1} \otimes \cdots \otimes \widetilde{\mathbf{D}}_{jm}) \;\; \textit{and } s \textit{ is the number of basic domains} \\
\varrho & \in & \mathbf{Env} & = \mathbf{Var} \to \widetilde{\mathcal{SD}} \\
\pi & \in & \mathbf{SigEnv} & = \mathbf{Fn} \to \widetilde{\mathcal{SD}}^{n+1} \\
\varsigma & \in & \mathbf{FEnv} & = \mathbf{Fn} \to \widetilde{\mathcal{SD}}^{n} \to \widetilde{\mathcal{SD}}
\end{array}$$

3. Valuation Functions

$$\begin{array}{lll}
\widetilde{\mathcal{M}} & : & \mathbf{Program} \to \widetilde{\mathcal{SD}}^{n} \to \mathbf{SigEnv} \\
\widetilde{\mathcal{E}} & : & \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Fenv} \to \widetilde{\mathcal{SD}} \\
\widetilde{\mathcal{A}} & : & \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Fenv} \to \mathbf{SigEnv}
\end{array}$$

$$\begin{array}{l}
\widetilde{\mathcal{M}}\, \llbracket \{f_i(x_1, \cdots, x_n) = e_i\} \rrbracket \, \langle \widetilde{\delta}_1, \cdots, \widetilde{\delta}_n \rangle = \widetilde{h}(\bot[\langle \widetilde{\delta}_1, \cdots, \widetilde{\delta}_n, \bot \rangle / \llbracket f_1 \rrbracket]) \\
\quad \textit{whererec} \quad \widetilde{h}\, \pi = \pi \sqcup \widetilde{h}(\bigsqcup\{\widetilde{\mathcal{A}}\, \llbracket e_i \rrbracket \, \bot[\widetilde{\delta}'_j / \llbracket x_j \rrbracket]\, \varsigma \mid \widetilde{\delta}'_j = (\pi\, f_i)\downarrow j \; ; \; \forall j \in \{1, \ldots, n\} \; ; \; \forall \llbracket f_i \rrbracket\}) \\
\qquad\qquad\qquad\; \varsigma = \bot[(\lambda(\widetilde{\delta}_1, \cdots, \widetilde{\delta}_n) \, . \, \widetilde{\mathcal{E}}\, \llbracket e_i \rrbracket \, \bot[\widetilde{\delta}_k / \llbracket x_k \rrbracket]\, \varsigma) / \llbracket f_i \rrbracket]
\end{array}$$

$$\begin{array}{lcl}
\widetilde{\mathcal{E}}\, \llbracket c \rrbracket \, \varrho\, \varsigma & = & \widetilde{\mathcal{K}}\, \llbracket c \rrbracket \\
\widetilde{\mathcal{E}}\, \llbracket x \rrbracket \, \varrho\, \varsigma & = & \varrho\, \llbracket x \rrbracket \\
\widetilde{\mathcal{E}}\, \llbracket p(e_1, \cdots, e_n) \rrbracket \, \varrho\, \varsigma & = & \widetilde{\mathcal{K}}_P\, \llbracket p \rrbracket \, (\widetilde{\mathcal{E}}\, \llbracket e_1 \rrbracket \, \varrho\, \varsigma) \cdots (\widetilde{\mathcal{E}}\, \llbracket e_n \rrbracket \, \varrho\, \varsigma) \\
\widetilde{\mathcal{E}}\, \llbracket if\ e_1\ e_2\ e_3 \rrbracket \, \varrho\, \varsigma & = & \widetilde{\delta}_1 = \bot_{\widetilde{\mathcal{SD}}} \to \bot_{\widetilde{\mathcal{SD}}}, \\
 & & \widetilde{\delta}_1^1 = \textit{Static} \to \widetilde{\delta}_2 \sqcup \widetilde{\delta}_3, \; \langle \textit{Dynamic}, \top_{\widetilde{\mathcal{D}}_2}, \ldots, \top_{\widetilde{\mathcal{D}}_m} \rangle \\
 & & \textit{where } \widetilde{\delta}_i = \widetilde{\mathcal{E}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma \quad \textit{for } i = \{1, 2, 3\} \\
\widetilde{\mathcal{E}}\, \llbracket f(e_1, \cdots, e_n) \rrbracket \, \varrho\, \varsigma & = & \exists j \in \{1, \ldots, n\} \; s.t. \; \widetilde{\delta}_j^1 = \textit{Dynamic} \to \langle \textit{Dynamic}, \top_{\widetilde{\mathcal{D}}_2}, \ldots, \top_{\widetilde{\mathcal{D}}_m} \rangle, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \varsigma\, \llbracket f \rrbracket \, (\widetilde{\delta}_1, \ldots, \widetilde{\delta}_n) \\
 & & \textit{where } \widetilde{\delta}_i = \widetilde{\mathcal{E}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma \quad \textit{for } i \in \{1, \ldots, n\}
\end{array}$$

$$\widetilde{\mathcal{K}}\, \llbracket c \rrbracket = \langle \widetilde{\Gamma}_1(d), \cdots, \widetilde{\Gamma}_m(d) \rangle \;\; \textit{where } \widetilde{\Gamma}_i = \widetilde{\alpha}_{\widetilde{\mathcal{D}}_i} \circ \widehat{\alpha}_{\widehat{\mathcal{D}}_i} \;\; \textit{and } d = (\mathcal{K}\; c)$$

$$\begin{array}{l}
\widetilde{\mathcal{K}}_P\, \llbracket p^c \rrbracket \, \widetilde{\delta}_1 \cdots \widetilde{\delta}_n = \widetilde{\omega}_{p^c}(\widetilde{\delta}_1, \cdots, \widetilde{\delta}_n) \quad \textit{where } p^c : \mathbf{D}^n \to \mathbf{D} \\
\widetilde{\mathcal{K}}_P\, \llbracket p^o \rrbracket \, \widetilde{\delta}_1 \cdots \widetilde{\delta}_n = \tilde{d} = \bot_{\widetilde{Values}} \to \bot_{\widetilde{\mathcal{SD}}}, \langle \tilde{d}, \top_{\widetilde{\mathcal{D}}'_2}, \cdots, \top_{\widetilde{\mathcal{D}}'_m} \rangle \\
\qquad\qquad\qquad\qquad\quad \textit{where } p^o : \mathbf{D}^n \to \mathbf{D}' \\
\qquad\qquad\qquad\qquad\qquad\quad \tilde{d} = \widetilde{\omega}_{p^o}(\widetilde{\delta}_1, \cdots, \widetilde{\delta}_n)
\end{array}$$

$$\begin{array}{lcl}
\widetilde{\mathcal{A}}\, \llbracket c \rrbracket \, \varrho\, \varsigma & = & \bot \\
\widetilde{\mathcal{A}}\, \llbracket x \rrbracket \, \varrho\, \varsigma & = & \bot \\
\widetilde{\mathcal{A}}\, \llbracket p(e_1, \cdots, e_n) \rrbracket \, \varrho\, \varsigma & = & \bigsqcup_{i=1}^{n} \widetilde{\mathcal{A}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma \\
\widetilde{\mathcal{A}}\, \llbracket if\ e_1\ e_2\ e_3 \rrbracket \, \varrho\, \varsigma & = & \bigsqcup_{i=1}^{3} \widetilde{\mathcal{A}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma \\
\widetilde{\mathcal{A}}\, \llbracket f(e_1, \cdots, e_n) \rrbracket \, \varrho\, \varsigma & = & (\bigsqcup_{i=1}^{n} \widetilde{\mathcal{A}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma) \sqcup \bot[\langle \widetilde{\delta}_1, \cdots, \widetilde{\delta}_n, \widetilde{\delta} \rangle / \llbracket f \rrbracket] \\
 & & \textit{where } \widetilde{\delta}_i = \widetilde{\mathcal{E}}\, \llbracket e_i \rrbracket \, \varrho\, \varsigma \quad \textit{for } i = \{1, \cdots, n\} \\
 & & \qquad\quad \widetilde{\delta} = \widetilde{\mathcal{E}}\, \llbracket f(e_1, \ldots, e_n) \rrbracket \, \varrho\, \varsigma
\end{array}$$

Fig. 4 Facet analysis.

```
iprod(A,B) =                  dotProd(A,B,n) =
  let n = Vec#(A)               if n = 0 then 0
   in dotProd(A,B,n)               else Vref(A,n) * Vref(B,n)
                                        + dotProd(A,B,n-1)
```

Fig. 5. Program for inner product computation.

$$Vec\sharp : \mathbf{V} \rightarrow \mathbf{Int}$$
$Vec\sharp$ returns the size of the vector
$$Vref : \mathbf{V} \times \mathbf{Int} \rightarrow \mathbf{Float}$$
$Vref$ returns a specified element of a given vector

The program for computing inner product is presented in Figure 5. To specialize the inner product program with respect to the size of the vectors our strategy consists of defining the size information as a property of a vector.

6.1 On-line Parameterized Partial Evaluation

In order to capture the size property of a vector, we define the **Size** facet $[\hat{\mathbf{V}}; \hat{\mathbf{O}}]$ from the vector algebra $[\mathbf{V}; \mathbf{O}]$.

(1) $\hat{\mathbf{V}} = \mathbf{Int} \cup \{\bot_{\hat{V}}, \top_{\hat{V}}\}$ with the ordering $\bot_{\hat{V}} \sqsubseteq i \sqsubseteq \top_{\hat{V}} \;\; \forall i \in \mathbf{Int}$.

(2) Abstraction function

$$\begin{array}{lcll} \hat{\alpha}_V & : & \mathbf{V} \rightarrow \hat{\mathbf{V}} & \\ \hat{\alpha}_{\hat{V}}(v) & = & \bot_{\hat{V}} & \text{if } v = \bot \\ & & \text{Vec}\sharp(v) & \text{otherwise} \end{array}$$

(3) Closed operators

$$\widehat{MkVec}: \widehat{\mathbf{Values}} \rightarrow \hat{\mathbf{V}}$$
$$\begin{aligned} \widehat{MkVec}(i) = {} & (i = \bot_{\widehat{Values}}) \rightarrow \bot_{\hat{V}}, \\ & (i = \top_{\widehat{Values}}) \rightarrow \top_{\hat{V}}, i \end{aligned}$$
$$\widehat{UpdVec}: \hat{\mathbf{V}} \times \widehat{\mathbf{Values}} \times \widehat{\mathbf{Values}} \rightarrow \hat{\mathbf{V}}$$
$$\widehat{UpdVec}(\hat{v}, i, r) = (i = \bot_{\widehat{Values}}) \vee (r = \bot_{\widehat{Values}}).$$

(4) Open operators

$$\widehat{Vec\sharp}: \hat{\mathbf{V}} \rightarrow \widehat{\mathbf{Values}}$$
$$\begin{aligned} \widehat{Vec\sharp}(\hat{v}) = {} & (\hat{v} = \bot_{\hat{V}}) \rightarrow \bot_{\widehat{Values}}, \\ & (\hat{v} = i) \rightarrow i, \top_{\widehat{Values}} \end{aligned}$$
$$\widehat{Vref}: \hat{\mathbf{V}} \times \widehat{\mathbf{Values}} \rightarrow \widehat{\mathbf{Values}}$$
$$\widehat{Vref}(\hat{v}, i) = (\hat{v} = \bot_{\hat{V}}) \vee (i = \bot_{\widehat{Values}}) \rightarrow \bot_{\widehat{Values}}, \top_{\widehat{Values}}$$

Let us now specialize the inner product program with respect to a given size, say 3. The facet values passed to the partial evaluator will be $\langle$A, $\langle \top_{\widehat{Values}}, 3\rangle\rangle$ and $\langle$B, $\langle \top_{\widehat{Values}}, 3\rangle\rangle$, (where A and B are residual identifiers for iprod). When partially evaluating iprod, the size facet information is used to obtain the size of vector A. Variable n is then bound to a constant value. As a result, the test expression in dotProd is static, and thus can be reduced; also, the recursive call to `dotProd` can be unfolded. The resulting program is displayed in Figure 6. Notice that it is now nonrecursive; also, since elements of the vectors are unknown at partial-evaluation time, the primitive operation `Vref` cannot be reduced; thus both the multiplication and addition operations are residual.

6.2 Off-line Parameterized Partial Evaluation

In the off-line parameterized partial evaluation, we define the abstract **Size** facet $[\tilde{\mathbf{V}}; \tilde{\mathbf{O}}]$.

(1) $\tilde{\mathbf{V}} = \{s, d\}$ with the ordering $\bot_{\tilde{V}} \sqsubseteq s \sqsubseteq d$.
Values s and d denote a static and a dynamic vector size, respectively.

(2) Abstraction function

$$\begin{array}{lll} \tilde{\alpha}_{\tilde{V}} : \hat{\mathbf{V}} \to \tilde{\mathbf{V}} & & \\ \tilde{\alpha}_{\tilde{V}}(\hat{v}) = \bot_{\tilde{V}} & \text{if } \hat{v} = \bot_{\hat{V}} \\ \qquad d & \text{if } \hat{v} = \top_{\hat{V}} \\ \qquad s & \text{otherwise.} \end{array}$$

(3) Closed operators

$\widetilde{MkVec}: \widetilde{\mathbf{Values}} \to \tilde{\mathbf{V}}$

$\widetilde{MkVec}(x) = (i = \bot_{\widetilde{Values}}) \to \bot_{\tilde{V}}\ , (i = \text{Dynamic}) \to d, s$

$\widetilde{UpdVec}: \tilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \times \widetilde{\mathbf{Values}} \to \hat{\mathbf{V}}$

$\widetilde{UpdVec}(\tilde{v}, i, r) = (i = \bot_{\widetilde{Values}}) \vee (r = \bot_{\widetilde{Values}}) \to \bot_{\tilde{V}}\ , \tilde{v}.$

(4) Open operators

$\widetilde{Vec\sharp}: \tilde{\mathbf{V}} \to \widetilde{\mathbf{Values}}$

$\widetilde{Vec\sharp}(\tilde{v}) = (\tilde{v} = \bot_{\tilde{V}}\ ) \to \bot_{\widetilde{Values}}, (\tilde{v} = s) \to \text{Static}, \text{Dynamic}$

$\widetilde{Vref}: \tilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \to \widetilde{\mathbf{Values}}$

$\widetilde{Vref}(\tilde{v}, i) = (\tilde{v} = \bot_{\tilde{V}}\ ) \vee (i = \bot_{\widetilde{Values}}) \to \bot_{\widetilde{Values}}, \text{Dynamic}$

Let us now perform a facet analysis on the inner product program, given that the actual value of both vectors is dynamic but their size is static. Recall that besides the abstract `Size` facet, the binding-time facet (Definition 10) is also defined. Both parameters of `dotProd` will then be bound to the pair of abstract values $\langle$`Dynamic, s`$\rangle$. As a result, the binding-time value of variable `n` is `Static`. Thus, the facet analysis determines that the test expression in `dotProd` is static, and the conditional expression can be reduced statically. This coincides with the result of on-line parameterized partial evaluation; however, these reductions have been determined statically.

Figure 7 displays the information yielded by the facet analysis of the inner product program when only the size of the vectors is static; more precisely, we show the facet values of the main expressions of the program. For conciseness, the values `Static` and `Dynamic` are noted `Stat` and `Dyn`, respectively.

The underlined binding-time value represents the static value obtained from the size abstract facet value. Notice that the size information is only

```
iprod(A,B) = Vref(A,3) * Vref(B,3)
            +Vref(A,2) * Vref(B,2)
            +Vref(A,1) * Vref(B,1)
```

Fig. 6. Residual program for inner product computation.

```
Program Code

iprod(A, B) =                          Facet Values
  let n = Vec♯(A)
  in dotProd(A, B, n)                  A = ⟨Dyn, s⟩, B = ⟨Dyn, s⟩
                                       Vec♯(A) = ⟨Stat⟩
dotProd(A, B, n) =                     n = ⟨Stat⟩
  if
    n = 0                              A = ⟨Dyn, s⟩, B = ⟨Dyn, s⟩
    then 0
    else vref(A, n) * Vref(B, n)       n = ⟨Stat⟩
              +                        ⟨Stat⟩
         dotProd(A, B, n − 1)          vref(A, n) = ⟨Dyn⟩, Vref(B, n) = ⟨Dyn⟩
```

Fig. 7. Abstract facet information after facet analysis.

used in the main function, iprod. This means that, at specialization time, size facet computation is only required for iprod (in fact, it is only required for partial evaluation of the abstract syntax tree rooted by the open operation Vec♯). Only partial-evaluation facet computation is performed for dotProd.

This contrasts with on-line parameterized partial evaluation of the inner product program where the size facet computation is performed each time function dotProd is invoked.

7. RELATED WORK

Redfun was the first partial evaluator to specialize programs with respect to symbolic values. Since then, other partial-evaluation systems with similar capabilities have been developed (e.g., [21, 13, 3]). The latest system developed along this line is Fuse [26, 25]. This partial evaluator utilizes *type information* during program specialization. In particular, it uses symbolic values to represent both a value and the code to produce the value. This technique is similar to value descriptors and q-tuples introduced in Redfun. However, instead of fixing the set of static properties used by the system (as in Redfun), Fuse allows the user to modify the specializer's code to introduce new type information.

In contrast to our approach, Fuse and its predecessors are restricted to an on-line strategy. They do not provide a safe and systematic approach to introduce user-defined static properties; when static properties can be introduced, this is usually done by modifying the partial evaluator's code. Finally, the lack of a formal methodology makes it difficult to reason about combining various symbolic values.

Generalized Partial Computation [11, 24] is a program optimization approach that aims at specializing programs with respect to different kinds of information (called *u-information*) such as logical structures of programs, axioms for abstract data types, and so on. It uses a set of transformation rules to specialize a program and calls upon a logic system to compute u-information for each expression in the program. The safety of the computation relies on the underlying logic system and its relation with the semantics of the language in which programs are written. This relation is currently under study [24]. Generalized Partial Computation does not address off-line partial evaluation. The literature in the field [23] does not report any implementation of this approach.

8. CONCLUSION AND FUTURE WORKS

Parameterized partial evaluation is based on a uniform methodology to introduce static properties at both the on-line and off-line levels of partial evaluation. This is achieved by the notion of facet mapping, which provides a generic and safe abstraction mechanism for relating domains and primitive operations at three levels of evaluation: standard evaluation, on-line partial evaluation, and off-line partial evaluation. Furthermore, our approach generalizes the notion of binding-time analysis to any facet information. Facet analysis makes it possible to achieve self-application and improve the specialization process.

Parameterized partial evaluation has already been successfully implemented for a first order subset of ML at CMU [6] and at Yale [17].

Both on-line and off-line higher order parameterized partial evaluation have also been formally specified. Not surprisingly, these specifications have essentially the same structure as higher order partial evaluators. In particular, in the case of off-line partial evaluation, we have directly used the binding-time analysis described in [8] and parameterized it with respect to abstract facets. We are now using these specifications as a basis for implementing a higher order parameterized partial evaluator for ML.

Finally, we are investigating various extensions to this framework. In particular, we are looking into parameterized partial evaluation for a lazy language. We are also exploring partial evaluation parameterized with respect to operational properties such as strictness properties.

APPENDIX A. PROOFS

A.1 Proofs on Logical Relations

PROPERTY 1. *Let* $\alpha_{A'}$: $[\mathbf{A};\mathbf{O}] \to [\mathbf{A}';\mathbf{O}']$ *be a facet mapping with respect to* $\alpha' = \{\alpha_{B_i'}: \mathbf{B}_i \to \mathbf{B}_i'\}$, $\forall p \in \mathbf{O}$ *and its corresponding abstract version* $p' \in \mathbf{O}'$, $p \sqsubseteq_{\alpha_{A'}} p'$.

PROOF. We need to prove that the safety condition (Condition 5) in Definition 2 is equivalent to the relation $p \sqsubseteq_{\alpha_{A'}} p'$ $\forall p \in \mathbf{O}$. We only prove the case for closed operator. The proof for open operator is similar, and thus omitted.

(1) Suppose that $\alpha_{A'} \circ p \sqsubseteq p' \circ \alpha_{A'}$. $\forall a \in \mathbf{A}$ and $\forall a' \in \mathbf{A}'$, if $a \sqsubseteq_{\alpha_{A'}} a'$, then

$$\begin{aligned} \alpha_{A'}(p(a)) &\sqsubseteq_{A'} p'(\alpha_{A'}(a)) && \text{by the above assumption} \\ &\sqsubseteq_{A'} p'(a') && \text{monotonicity of } p' \text{ and } a \sqsubseteq_{\alpha_{A'}} a'. \end{aligned}$$

Thus, $p(a) \sqsubseteq_{\alpha_{A'}} p'(a')$. Since this is true for any $a \in \mathbf{A}$ and $a' \in \mathbf{A}'$ with $a \sqsubseteq_{\alpha_{A'}} a'$, we have $p \sqsubseteq_{\alpha_{A'}} p'$.

(2) Suppose that $p \sqsubseteq_{\alpha_{A'}} p'$. $\forall a \in \mathbf{A}$,

$$\begin{aligned} p \sqsubseteq_{\alpha_{A'}} p' &\Rightarrow p(a) \sqsubseteq_{\alpha_{A'}} p(\alpha_{A'}(a)) && \text{since } a \sqsubseteq_{\alpha_{A'}} \alpha_{A'}(a) \\ &\Leftrightarrow \alpha_{A'}(p(a)) \sqsubseteq_{A'} p(\alpha_{A'}(a)) && \text{by Definition 3} \\ &\Rightarrow (\alpha_{A'} \circ p)(a) \sqsubseteq (p' \circ \alpha_{A'})(a). \end{aligned}$$

Since this is true for all $a \in \mathbf{A}$, we have $\alpha_{\mathbf{A}'} \circ p \sqsubseteq p' \circ \alpha_{A'}$.

This concludes the proof. □

A.2 Proofs on On-line Parameterized Partial Evaluation

PROPERTY 2. *For any open operator $p \in \mathbf{O}$ of arity n, $\forall \hat{d}_1, \cdots, \hat{d}_n \in \hat{\mathbf{D}}$ and $\forall d_i \in \mathbf{D}$, if $d_i \sqsubseteq_{\hat{\alpha}_{\hat{\mathcal{D}}}} \hat{d}_i$ $\forall i \in \{1, \cdots, n\}$, then $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \in$ **Const** $\wedge$ $p(d_1, \cdots, d_n) \neq \bot \Rightarrow \hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \hat{\tau}(p(d_1, \cdots, d_n))$.*

PROOF. Let $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = c_1$ and $\hat{\tau}(p(d_1, \cdots, d_n)) = c_2$, where $c_1, c_2 \in \widehat{\mathbf{Values}}$. Notice that $c_2 \neq \bot$. By the safety condition for facet, we must have $c_2 \sqsubseteq_{\widehat{Values}} c_1$. Since any two distinct constants are incomparable in $\widehat{\mathbf{Values}}$, we must have $c_1 = c_2$. □

LEMMA 3. *Let $[\hat{\mathcal{D}}; \hat{\Omega}]$ be a product of facets and $p \in \mathbf{O}$ be an open operator. If $\exists j, k \in \{1, \cdots, m\}$ $(j \neq k)$ and $\hat{\delta}_1, \cdots, \hat{\delta}_n \in \hat{\mathcal{D}}$ such that both $\hat{p}_j(\hat{\delta}_1^j, \cdots, \hat{\delta}_n^j) \in$ **Const** and $\hat{p}_k(\hat{\delta}_1^k, \cdots, \hat{\delta}_n^k) \in$ **Const**, then $\hat{p}_j(\hat{\delta}_1^j, \cdots, \hat{\delta}_n^j) = \hat{p}_k(\hat{\delta}_1^k, \cdots, \hat{\delta}_n^k)$.*

PROOF. Without loss of generality, we consider unary open operators (the argument is noted $\hat{\delta}$). Let $C = \bigcap_{i=1}^{m} \{d \in \mathbf{D} \mid d \sqsubseteq_{\hat{\alpha}_i} \hat{\delta}^i\}$. Since $\hat{\delta}$ is consistent, it is true that $C \neq \emptyset$ and $C \neq \{\bot\}$. Suppose $\exists d \in C$ such that $p(d)$ terminates. Then, by Property 2, we have

$$\hat{p}_j(\hat{\delta}^j) \in \mathbf{Const} \Rightarrow \hat{p}_j(\hat{\delta}^j) = \hat{\tau}(p(d)), \quad \text{and}$$

$$\hat{p}_k(\hat{\delta}^k) \in \mathbf{Const} \Rightarrow \hat{p}_k(\hat{\delta}^k) = \hat{\tau}(p(d)).$$

Thus, $\hat{p}_j(\hat{\delta}^j) = \hat{\tau}(p(d)) = \hat{p}_k(\hat{\delta}^k)$. □

PROPERTY 4. *All operators defined in the product of facets, $[\hat{\mathcal{D}}; \hat{\Omega}]$, are monotonic.*

PROOF. It is easy to see that the operator for closed operation is monotonic, since all its constituent facet operations are monotonic.

To prove that function $\hat{\omega}_p$ for open operations is indeed monotonic, we first observe the fact that $\forall \hat{\delta}_1, \cdots, \hat{\delta}_n \in \hat{\mathscr{D}}$, $\forall j \in \{1, \ldots, m\}$:

$$\hat{\omega}_p\left(\hat{\delta}_1, \cdots, \hat{\delta}_n\right) \sqsubseteq_{\widehat{Values}} \hat{u}_j \tag{1}$$

where $\hat{u} = \langle \hat{p}_1(\hat{\delta}_1^1, \cdots, \hat{\delta}_n^1), \ldots, \hat{p}_m(\hat{\delta}_1^m, \cdots, \hat{\delta}_n^m)\rangle$. Without loss of generality, we assume that the operator takes one argument. Thus, we need to show that $\forall \hat{\delta}_1, \hat{\delta}_2 \in \hat{\mathscr{D}}$,

$$\hat{\delta}_1 \sqsubseteq_{\hat{\mathscr{D}}} \hat{\delta}_2 \Rightarrow \hat{\omega}_p\left(\hat{\delta}_1\right) \sqsubseteq_{\widehat{Values}} \hat{\omega}_p\left(\hat{\delta}_2\right).$$

Let $\hat{v} = \hat{\omega}_p(\hat{\delta}_1)$; the proof is done by case analysis of the different classes of value $\hat{v}$ produced by the operation.

(1) If $\hat{v} = \bot_{\widehat{Values}}$, then $\hat{v} \sqsubseteq_{\widehat{Values}} \hat{\omega}_p(\hat{\delta}_2)$ since $\hat{v}$ is the least element in $\widehat{\mathbf{Values}}$.

(2) If $\hat{v} \in \mathbf{Const}$, then by (1), $\forall j \in \{1, \ldots, m\}$, we have

$$\bot_{\widehat{Values}} \sqsubset_{\widehat{Values}} \hat{v} \sqsubseteq_{\widehat{Values}} \hat{p}_j\left(\hat{\delta}_1^j\right) \sqsubseteq_{\widehat{Values}} \hat{p}_j\left(\hat{\delta}_2^j\right).$$

If $\exists k \in \{1, \ldots, m\}$ such that $\hat{p}_k(\hat{\delta}_2^k)$ is constant, then this constant must be $\hat{v}$ and $\hat{v} = \hat{\omega}_p(\hat{\delta}_2)$; otherwise, $\hat{v} \sqsubseteq_{\widehat{Values}} \hat{\omega}_p(\hat{\delta}_2) = \top_{\widehat{Values}}$.

(3) If $\hat{v} = \top_{\widehat{Values}}$, then by (1), $\forall j \in \{1, \ldots, m\}$, we have

$$\top_{\widehat{Values}} \sqsubseteq_{\widehat{Values}} \hat{p}_j\left(\hat{\delta}_1^j\right) \sqsubseteq_{\widehat{Values}} \hat{p}_j\left(\hat{\delta}_2^j\right).$$

But $\top_{\widehat{Values}}$ is the top element in $\widehat{\mathbf{Values}}$, therefore, $\forall j \in \{1, \ldots, m\}$, $\hat{p}_j(\hat{\delta}_2^j) = \top_{\widehat{Values}}$, and $\hat{\omega}_p(\hat{\delta}_2) = \hat{v}$.

Therefore, $\hat{\omega}_p$ is monotonic. □

PROPERTY 5. *The partial-evaluation facet (Definition 7) is a facet.*

PROOF. We need to show that $\alpha_{\widehat{Values}}: [\mathbf{D}; \mathbf{O}] \to [\widehat{\mathbf{Values}}; \hat{\mathbf{O}}]$ is a facet mapping with respect to $\hat{\tau}$. This is accomplished by considering the conditions for a facet mapping.

(1) $\widehat{\mathbf{Values}}$ is an algebraic lattice of height 3.

(2) We want to show that $\forall \hat{p} \in \hat{\mathbf{O}}$, $\hat{p}$ is monotonic. Without loss of generality, we assume that $\hat{p}$ takes one argument. Thus, we need to show that

$$\forall \hat{d}_1, \hat{d}_2, \hat{d}_1 \sqsubseteq \hat{d}_2 \Rightarrow \hat{p}\left(\hat{d}_1\right) \sqsubseteq \hat{p}\left(\hat{d}_2\right).$$

The proof is done by case analysis of the different values of $\hat{d}_2$.

—$\hat{d}_2 = \bot_{\widehat{Values}}$. Then $\hat{d}_1 = \bot_{\widehat{Values}}$ too. By the definition of $\hat{p}$, we have $\hat{p}(\hat{d}_1) = \bot_{\widehat{Values}} = \hat{p}(\hat{d}_2)$.

—$\hat{d}_2 \in \mathbf{Const}$. Then either $\hat{d}_1 = \hat{d}_2$ or $\hat{d}_1 = \bot_{\widehat{Values}}$. For the former case, we have $\hat{p}(\hat{d}_1) = \hat{p}(\hat{d}_2)$. For the latter case, we have $\hat{p}(\hat{d}_1) = \bot_{\widehat{Values}}$.

Since $\bot_{\widehat{Values}}$ is the least element in the domain, then $\hat{p}(\hat{d}_1) \sqsubseteq \hat{p}(\hat{d}_2)$.
—$\hat{d}_2 = \top_{\widehat{Values}}$. Then $\hat{p}(\hat{d}_2) = \top_{\widehat{Values}}$ by definition of $\hat{p}$. Since $\top_{\widehat{Values}}$ is the maximal element in the domain, $\hat{p}(\hat{d}_1) \sqsubseteq \hat{p}(\hat{d}_2)$.

Hence, $\forall \hat{p} \in \hat{\mathbf{O}}$, $\hat{p}$ is monotonic.

(3) If $p \in \mathbf{O}$ is a closed operator, then its corresponding abstract version is p': $\widehat{\mathbf{Values}} \to \widehat{\mathbf{Values}}$, whose type matches that of $\hat{p}$ as defined in Definition 7.

(4) If $p \in \mathbf{O}$ is an open operator with functionality $\mathbf{D} \to \mathbf{D}'$, where $\mathbf{D}'$ is some domain different from $\mathbf{D}$, then its corresponding abstract version is p': $\widehat{\mathbf{Values}} \to \widehat{\mathbf{Values}}$, whose type again matches that of $\hat{p}$ in Definition 7.

(5) To prove the safety of the abstract operations, we define $\sqsubseteq_{\alpha \widehat{Values}}$ as a relation between $\mathbf{D}$ and $\widehat{\mathbf{Values}}$ such that:

$$\forall d \in \mathbf{D}, \forall \hat{d} \in \widehat{\mathbf{Values}}: d \sqsubseteq_{\alpha \widehat{Values}} \hat{d} \Leftrightarrow \alpha_{\widehat{Values}}(d) \sqsubseteq_{\widehat{Values}} \hat{d}.$$

We need to show that $\forall p \in \mathbf{O}$, if $\hat{p} \in \hat{\mathbf{O}}$ is its corresponding abstract version, then $p \sqsubseteq_{\alpha \widehat{Values}} \hat{p}$. That is, $\forall i \in \{1, \ldots, n\}$, $\forall d_i \in \mathbf{D}$, $\forall \hat{d}_i \in \widehat{\mathbf{Values}}$:

$$\bigwedge_{i=1}^{n} \left(d_i \sqsubseteq_{\alpha \widehat{Values}} \hat{d}_i\right) \Rightarrow p(d_1, \cdots, d_n) \sqsubseteq_{\alpha \widehat{Values}} \hat{p}\left(\hat{d}_1, \cdots, \hat{d}_n\right). \tag{2}$$

This is achieved by considering the result produced by $p(d_1, \cdots, d_n)$. Notice that $p(d_1, \ldots, d_n)$ returns either a concrete value or $\bot$.

(a) If $p(d_1, \ldots, d_n) = \bot$, then (2) is vacuously true.

(b) Suppose that $p(d_1, \cdots, d_n) \neq \bot$, then $\forall i \in \{1, \ldots, n\}$, $d_i \neq \bot$ since p is strict. This implies that $\hat{d}_i \neq \bot$, assuming that $d_i \sqsubseteq_{\alpha \widehat{Values}} \hat{d}_i$. It also implies that $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \neq \bot$; so $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$ is either a constant or it is $\top_{\widehat{Values}}$. If $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \top_{\widehat{Values}}$, then $p(d_1, \cdots, d_n) \sqsubseteq_{\alpha \widehat{Values}} \hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$. If $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$ produces a constant, then $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \hat{\tau}(p(d_1, \cdots, d_n))$ by the definition of $\hat{p}$. Again, $p(d_1, \cdots, d_n) \sqsubseteq_{\alpha \widehat{Values}} \hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$. Therefore, Definition 7 holds.

Thus, $\alpha_{\widehat{Values}}$ is a facet mapping with respect to $\widehat{\mathbf{Values}}$, and Definition 7 defines a facet. □

THEOREM 1. *Let $[\hat{\mathscr{D}}; \hat{\Omega}]$ be a product of facets (including the partial-evaluation facet) for an algebra* $[\mathbf{D}; \mathbf{O}]$. *Let* $c = (\mathscr{P}\mathscr{E}[\![p(x_1, \cdots, x_n)]\!] \bot [\langle [\![x_i]\!], \hat{\delta}_i \rangle / [\![x_i]\!]] \bot_{FnEnv} \bot_{Sf}) \downarrow 1$, *and* $v = (\mathscr{E}[\![p(x_1, \cdots, x_n)]\!] \bot [d_i / [\![x_i]\!]] \bot_{FnEnv})$ *where* $d_i \in \bigcap_{j=1}^{m} \{d \in \mathbf{D} \mid d \sqsubseteq_{\hat{\alpha}_{\mathscr{D}_j}} \hat{\delta}_i^j\}$, *for* $i \in \{1, \ldots, n\}$. *Then,*

$$(c \in \mathbf{Const}) \textit{ and } v \neq \bot \Rightarrow c = \hat{\tau}(v).$$

PROOF. First of all notice that

$$\left(\mathscr{E}[\![p(x_1, \cdots, x_n)]\!] \bot \left[d_i / [\![x_i]\!]\right] \bot\right) = \mathscr{K}_P[\![p]\!](d_1, \cdots, d_n) = p(d_1, \cdots, d_n).$$

As defined in the on-line parameterized partial-evaluation semantics, we have

$$\left(\mathscr{PE}[\![p(x_1,\cdots,x_n)]\!]\bot\left[\langle[\![x_\iota]\!],\hat{\delta}_\iota\rangle/[\![x_\iota]\!]\right]\bot\bot\right)\downarrow 1$$
$$=\left(\hat{\mathscr{K}}_P[\![p]\!]\langle[\![x_1]\!],\cdots,[\![x_n]\!]\rangle\langle\hat{\delta}_1,\cdots,\hat{\delta}_n\rangle\bot\right)\downarrow 1.$$

Given that $(\mathscr{PE}[\![p(x_1,\cdots,x_n)]\!]\bot[\langle[\![x_\iota]\!],\hat{\delta}_\iota\rangle/[\![x_\iota]\!]]\bot\bot)\downarrow 1\in$ **Const**, the proof is done by case analysis of the different classes of primitive operators:

(1) If p is a closed operator: Given $(\hat{\mathscr{K}}_P[\![p]\!]\langle[\![x_1]\!],\cdots,[\![x_n]\!]\rangle\langle\hat{\delta}_1,\cdots,\hat{\delta}_n\rangle\bot)\downarrow 1$ $\in$ **Const**. By definition of parameterized online partial evaluation, for a closed operator, this constant can only be produced by the partial-evaluation facet (that is, the first component of the product of facets). Thus,

$$\left(\hat{\mathscr{K}}_P[\![p]\!]\langle[\![x_1]\!],\cdots,[\![x_n]\!]\rangle\langle\hat{\delta}_1,\cdots,\hat{\delta}_n\rangle\bot\right)\downarrow 1=\hat{p}_1\left(\hat{\delta}_1^1,\cdots,\hat{\delta}_n^1\right)\in\mathbf{Const}.$$

Given that $(\mathscr{E}[\![p(x_1,\cdots,x_n)]\!]\bot[d_\iota/[\![x_\iota]\!]]\bot)\neq\bot$ and $\hat{p}_1(\hat{\delta}_1^1,\cdots,\hat{\delta}_n^1)\in$ **Const**, by Definition 7, we have $\forall i\in\{1,\cdots,n\}$, $\hat{\delta}_\iota^1\in$ **Const**. Then $\forall i\in\{1,\cdots,n\}$, $\forall d_\iota\in\mathbf{D}$ such that $d_\iota=\mathscr{K}(\hat{\delta}_\iota^1)$:

$$\hat{p}_1\left(\hat{\delta}_1^1,\cdots,\hat{\delta}_n^1\right)=\hat{\tau}\left(\mathscr{K}_P[\![p]\!](d_1,\cdots,d_n)\right)=\hat{\tau}(p(d_1,\cdots,d_n)).$$

(2) If p is an open operator: $(\hat{\mathscr{K}}_P[\![p]\!]\langle[\![x_1]\!],\cdots,[\![x_n]\!]\rangle\langle\hat{\delta}_1,\cdots,\hat{\delta}_n\rangle\bot)\downarrow 1\in$ **Const** implies that this constant is produced by a facet operation in the product of facets. Lemma 3 says that we can consider any facet that produces the constant. Assume that the ιth facet produces this constant; This is denoted by $\hat{p}_\iota(\hat{\delta}_1^\iota,\cdots,\hat{\delta}_n^\iota)$. By Property 2, we have $\hat{p}_\iota(\hat{\delta}_1^\iota,\cdots,\hat{\delta}_n^\iota)=\hat{\tau}(p(d_1,\cdots,d_n))$.

This concludes the proof. □

A.3 Proofs on Off-line Parameterized Partial Evaluation

Property 6. *For any open operator $\hat{p}\in\hat{\mathbf{O}}$ of arity n, $\forall\tilde{d}_\iota,\cdots,\tilde{d}_n\in\tilde{\mathbf{D}}$ and $\forall\hat{d}_\iota\in\hat{\mathbf{D}}$, if $\hat{d}_\iota\sqsubseteq_{\tilde{\alpha}_{\tilde{D}}}\tilde{d}_\iota$ for $i\in\{1,\cdots,n\}$, then $(\tilde{p}(\tilde{d}_1,\cdots,\tilde{d}_n)=Static)\Rightarrow\hat{p}(\hat{d}_1,\cdots,\hat{d}_n)\sqsubseteq_{\widehat{Values}}c$ with $c\in$ **Const**.*

Proof. By the safety condition for abstract facet, we must have

$$\tilde{\tau}\left(\hat{p}\left(\hat{d}_1,\cdots,\hat{d}_n\right)\right)\sqsubseteq_{\widetilde{Values}}\tilde{p}\left(\tilde{d}_1,\cdots,\tilde{d}_n\right)=Static.$$

By definition of $\tilde{\tau}$, we have $\forall x\in\widehat{\mathbf{Values}}$, $\tilde{\tau}(x)\sqsubseteq_{\widetilde{Values}}Static\Rightarrow x\in\mathbf{Const}\cup\{\bot_{\widehat{Values}}\}$. Therefore $\hat{p}(\hat{d}_1,\cdots,\hat{d}_n)\in\mathbf{Const}\cup\{\bot_{\widehat{Values}}\}$. □

Property 7. *All operators defined in the product of abstract facets, $[\tilde{\mathscr{D}};\tilde{\Omega}]$, are monotonic.*

Proof. It is easy to see that the operator for closed operation is monotonic, since all its constituent abstract-facet operations are monotonic.

To prove that operator for open operation, $\tilde{\omega}_p$, is indeed monotonic, we first observe the fact that $\forall \tilde{\delta}_1, \cdots, \tilde{\delta}_n \in \tilde{\mathscr{D}}$ $\forall j \in \{1, \ldots, m\}$:

$$\tilde{\omega}_p\left(\tilde{\delta}_1, \cdots, \tilde{\delta}_n\right) \sqsubseteq_{\widetilde{Values}} \tilde{u}_j \tag{3}$$

where $\tilde{u} = \langle \tilde{p}_1(\tilde{\delta}_1^1, \cdots, \tilde{\delta}_n^1), \ldots, \tilde{p}_m(\tilde{\delta}_1^m, \cdots, \tilde{\delta}_n^m)\rangle$. Without loss of generality, we assume that the operator takes one argument. Thus, we need to show that $\forall \tilde{\delta}_1, \tilde{\delta}_2 \in \tilde{\mathscr{D}}$,

$$\tilde{\delta}_1 \sqsubseteq_{\tilde{\mathscr{D}}} \tilde{\delta}_2 \Rightarrow \tilde{\omega}_p\left(\tilde{\delta}_1\right) \sqsubseteq_{\widetilde{Values}} \tilde{\omega}_p\left(\tilde{\delta}_2\right).$$

Let $\tilde{v} = \tilde{\omega}_p(\tilde{\delta}_1)$, the proof is done by case analysis of the different classes of value $\tilde{v}$ produced by the operation.

(1) If $\tilde{v} = \bot_{\widetilde{Values}}$, then $\tilde{v} \sqsubseteq_{\widetilde{Values}} \tilde{\omega}_p(\tilde{\delta}_2)$ since $\tilde{v}$ is the least element in $\widetilde{\mathbf{Values}}$.

(2) If $\tilde{v} = \mathit{Static}$, then by (3), $\forall j \in \{1, \ldots, m\}$, we have

$$\bot_{\widetilde{Values}} \sqcup_{\widetilde{Values}} \tilde{v} \sqsubseteq_{\widetilde{Values}} \tilde{p}_j\left(\tilde{\delta}_1^j\right) \sqsubseteq_{\widetilde{Values}} \tilde{p}_j\left(\tilde{\delta}_2^j\right).$$

If $\exists k \in \{1, \ldots, m\}$ such that $\tilde{p}_k(\tilde{\delta}_2^k) = \mathit{Static}$, then by the definition of the product operator, $\tilde{v} = \tilde{\omega}_p(\tilde{\delta}_2)$; otherwise, $\tilde{v} \sqsubseteq_{\widetilde{Values}} \tilde{\omega}_p(\tilde{\delta}_2) = \mathit{Dynamic}$.

(3) If $\tilde{v} = \mathit{Dynamic}$, then by (3), $\forall j \in \{1, \ldots, m\}$, we have

$$\mathit{Dynamic} = \tilde{v} \sqsubseteq_{\widetilde{Values}} \tilde{p}_j\left(\tilde{\delta}_1^j\right) \sqsubseteq_{\widetilde{Values}} \tilde{p}_j\left(\tilde{\delta}_2^j\right).$$

But $\mathit{Dynamic}$ is the top element in $\widetilde{\mathbf{Values}}$; therefore, $\forall j \in \{1, \ldots, m\}$, $\tilde{p}_j(\tilde{\delta}_2^j) = \mathit{Dynamic}$, and $\tilde{\omega}_p(\tilde{\delta}_2) = \tilde{v}$. Therefore, $\tilde{\omega}_p$ is monotonic. □

PROPERTY 8. *The binding-time facet (Definition 10) is an abstract facet.*

PROOF. We need to show that $\tilde{\alpha}_{\widetilde{Values}}$: $[\widetilde{\mathbf{Values}}; \hat{\mathbf{O}}] \to [\widetilde{\mathbf{Values}}; \tilde{\mathbf{O}}]$ is a facet mapping with respect to $\tilde{\tau}$. This is accomplished by considering the conditions for a facet mapping.

(1) $\widetilde{\mathbf{Values}}$ is an algebraic lattice of height 3.

(2) We want to show that $\forall \tilde{p} \in \tilde{\mathbf{O}}$, $\tilde{p}$ is monotonic. Without loss of generality, we assume that $\tilde{p}$ takes one argument. Thus, we need to show that

$$\forall \tilde{d}_1, \tilde{d}_2, \tilde{d}_1 \sqsubseteq \tilde{d}_2 \Rightarrow \tilde{p}\left(\tilde{d}_1\right) \sqsubseteq \tilde{p}\left(\tilde{d}_2\right).$$

The proof is done by case analysis of the different values of $\tilde{d}_2$.

—$\tilde{d}_2 = \bot_{\widetilde{Values}}$. Then $\tilde{d}_1 = \bot_{\widetilde{Values}}$ too. By the definition of $\tilde{p}$, we have $\tilde{p}(\tilde{d}_1) = \bot_{\widetilde{Values}} = \tilde{p}(\tilde{d}_2)$.

—$\tilde{d}_2 = \mathit{Static}$. Then either $\tilde{d}_1 = \tilde{d}_2$ or $\tilde{d}_1 = \bot_{\widetilde{Values}}$. For the former case, we have $\tilde{p}(\tilde{d}_1) = \tilde{p}(\tilde{d}_2)$. For the latter case, we have $\tilde{p}(\tilde{d}_1) = \bot_{\widetilde{Values}}$. Since $\bot_{\widetilde{Values}}$ is the least element in the domain, thus, $\tilde{p}(\tilde{d}_1) \sqsubseteq \tilde{\boldsymbol{p}}(\tilde{\boldsymbol{d}}_2)$.

—$\tilde{d}_2 = \mathit{Dynamic}$. Then $\tilde{p}(\tilde{d}_2) = \mathit{Dynamic}$ by definition of $\tilde{p}$. Since $\mathit{Dynamic}$ is the maximal element in the domain. $\tilde{p}(\tilde{d}_1) \sqsubseteq \tilde{p}(\tilde{d}_2)$.

Hence, $\forall \tilde{p} \in \tilde{\mathbf{O}}$, $\tilde{p}$ is monotonic.

(3) If $\hat{p} \in \hat{\mathbf{O}}$ is a closed operator, then its corresponding abstract version is $\tilde{p}'$: $\widetilde{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}}$, the type of which matches that of $\tilde{p}$ in Definition 10.

(4) We do not need to consider open operator since $\hat{\mathbf{O}}$ has none.

(5) To prove the safety of the abstract operations, we define $\sqsubseteq_{\alpha \widetilde{Values}}$ as a relation between $\widehat{\mathbf{Values}}$ and $\widetilde{\mathbf{Values}}$ such that

$$\forall \hat{d} \in \widehat{\mathbf{Values}}, \forall \tilde{d} \in \widetilde{\mathbf{Values}}: \hat{d} \sqsubseteq_{\alpha \widetilde{Values}} \tilde{d} \Leftrightarrow \alpha_{\widetilde{Values}}(\hat{d}) \sqsubseteq_{\widetilde{Values}} \tilde{d}.$$

We need to show that $\forall \hat{p} \in \hat{\mathbf{O}}$, if $\tilde{p} \in \tilde{\mathbf{O}}$ is its corresponding abstract version, then $\hat{p} \sqsubseteq_{\alpha \widetilde{Values}} \tilde{p}$. That is, $\forall i \in \{1, \ldots, n\}$, $\forall \hat{d}_i \in \widehat{\mathbf{Values}}, \forall \tilde{d}_i \in \widetilde{\mathbf{Values}}$:

$$\bigwedge_{i=1}^{n} \left(\hat{d}_i \sqsubseteq_{\alpha \widetilde{Values}} \tilde{d}_i\right) \Rightarrow \hat{p}\left(\hat{d}_1, \cdots, \hat{d}_n\right) \sqsubseteq_{\alpha \widetilde{Values}} \tilde{p}\left(\tilde{d}_1, \cdots, \tilde{d}_n\right). \tag{4}$$

This is achieved by considering the result produced by $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$.

(a) If $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \bot_{\widehat{Values}}$, then (4) is vacuously true.

(b) If $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n)$ returns a constant, then by Definition 7, we have $\forall i \in \{1, \cdots, n\}$, $\hat{d}_i \in \mathbf{Const}$. This implies that $\forall i \in \{1, \cdots, n\}$, $\tilde{d}_i \in \{Static, Dynamic\}$, since $\hat{d}_i \sqsubseteq_{\alpha \widetilde{Values}} \tilde{d}_i$. From the definition of $\tilde{p}$, we must have $\tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n) \in \{Static, Dynamic\}$. Therefore, $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \sqsubseteq_{\alpha \widetilde{Values}} \tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n)$.

(c) If $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) = \top_{\widehat{Values}}$, then from the definition of $\hat{p}$ (Definition 7), $\exists j \in \{1, \cdots, n\}$ such that $\hat{d}_j = \top_{\widehat{Values}}$, while $\forall i \in \{1, \cdots, n\}$, $\hat{d}_i \neq \bot$. This implies that $\exists j \in \{1, \cdots, n\}$ such that $\tilde{d}_j = Dynamic$, while $\forall i \in \{1, \cdots, n\}$, $\tilde{d}_i \neq \bot$ (Since $\forall i \in \{1, \cdots, n\}$, $\hat{d}_i \sqsubseteq_{\alpha \widetilde{Values}} \tilde{d}_i$ in (4)). Thus, $\tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n) = \top_{\widetilde{Values}}$. Hence, $\hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \sqsubseteq_{\alpha \widetilde{Values}} \tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n)$.

Therefore, (4) holds.

Thus, $\alpha_{\widetilde{Values}}$ is a facet mapping with respect to $\tilde{\tau}$, and Definition 10 defines an abstract facet. □

ACKNOWLEDGMENTS

To the Yale Haskell Group. Thanks are also due to Karoline Malmkjær, Olivier Danvy, Paul Hudak, Pierre Jouvelot, and David Schmidt for thoughtful comments on earlier versions of this paper.

REFERENCES

1. ABRAMSKY, A. Abstract interpretation, logical relations and Kan extensions. *Logic Comput 1*, 1 (1990), 5–40.
2. ABRAMSKY, S., AND HANKIN, C., EDS. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
3. BERLIN, A. Partial evaluation applied to numerical computation In *ACM Conference on Lisp and Functional Programming* (Nice, France, June 1990), 139–150.

4. Bjørner, D., Ershov, A. P., and Jones, N. D., Eds. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
5. Burstall, R. M., and Darlington, J. A transformational system for developing recursive programs. *J. ACM 24*, 1 (1977), 44–67.
6. Colby, C., and Lee, P. An implementation of parameterized partial evaluation. *Bigre J. 74* (1991), 82–89.
7. Consel, C. Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs. Ph.D. thesis, Univ. de Paris VI, Paris, France, June 1989.
8. Consel, C. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming* (Nice, France, June 1990), 264–272.
9. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages* (Los Angeles, Calif., Jan. 1977), 238–252.
10. Emanuelson, P., and Haraldsson, A. On compiling embedded languages in LISP. In *ACM Conference on Lisp and Functional Programming* (Stanford, Calif., June 1980), 208–215.
11. Futamura, Y., and Nogi, K. Generalized partial computation. In *Partial Evaluation and Mixed Computation*. D. Bjørner, A. P. Ershov, and N. D. Jones, Eds. North-Holland, 1988.
12. Ganzinger, H., and Jones, N. D., Eds. *Programs as Data Objects, Vol. 217, Lecture Notes in Computer Science*, Springer-Verlag, 1985.
13. Guzowski, M. A. Toward developing a reflexive partial evaluator for an interesting subset of Lisp. Master's thesis, Dept. of Computer Engineering and Science, Case Western Reserve Univ., Cleveland, Ohio, 1988.
14. Haraldsson, A. A program manipulation system based on partial evaluation. Ph.D. thesis, Linköping Univ., Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.
15. Jones, N. D., and Nielson, F. Abstract interpretation: A semantics-based tool for program analysis. Tech. Rep. Univ. of Copenhagen and Aarhus Univ., Copenhagen, Denmark, 1990.
16. Jones, N. D., Sestoft, P., and Søndergaard, H. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp Symb. Comput. 2* (1989), 9–50.
17. Khoo, S. C. Parameterized partial evaluation: theory and practice. Ph.D. thesis, Yale Univ. 1992. June 1992. Also Res. Rep. 926.
18. Launchbury, J. Projection factorisation in partial evaluation. Ph.D. thesis, Dept. of Computing Science, Univ. of Glasgow, Scotland, 1990.
19. Nielson, F. Two-level semantics and abstract interpretation. *Theor. Comput. Sci. 69* (1989), 117–242.
20. Schmidt, D. A. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, 1986.
21. Schooler, R. Partial evaluation as a means of language extensibility. Master's thesis, MIT, 1984.
22. Sestoft, P. The structure of a self-applicable partial evaluator. In *Programs as Data Objects, Vol. 217, Lecture Notes in Computer Science*. Springer-Verlag, 1985, 236–256.
23. Sestoft, P. Annotated bibliography on partial evaluation and mixed computation. Diku report, Univ. of Copenhagen, Copenhagen, Denmark, 1990.
24. Takano, A. Generalized partial computation for a lazy functional language. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New Haven, Conn., June 1991), 1–11.
25. Weise, D., Conybeare, R., Ruf, E., and Seligman, S. Automatic online partial evaluation. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture* (Cambridge, Mass., Aug. 1991), 165–191.
26. Weise, D., and Ruf, E. Computing types during program specialization. Tech. Rep. 441, Stanford Univ., Stanford, Calif., 1990.

Received September 1991; revised March 1992; accepted April 1992.