Parameterized Partial Evaluation *

Charles Consel Siau Cheng Khoo

Yale University
Department of Computer Science
New Haven, CT 06520
{consel, khoo}@cs.yale.edu

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 [13] and carried out in practice with a system called Redfun in the late seventies. Although this work certainly started in the right direction, it has some limitations: (*i*) the static properties cannot be defined by the user; they are fixed; (*ii*) the approach is not formally defined: no safety condition for the definition of symbolic values, no finiteness criteria for fixpoint iteration, *etc.*; and (*iii*) because Redfun is an *online* partial evaluator — the treatment of the 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, 13], 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. Using abstract interpretation [1, 16], this can be formally achieved by relating the 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, considering partial evaluation as an algebra whose domain is syntactic terms and operations are primitive functions, it is possible to capture the partial evaluation itself as a facet.

*This 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.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

In "conventional" partial evaluation [3], efficiency is achieved by an *offline* 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 *offline* partial evaluator enables realistic self-application [17].

Our framework is general enough to capture offline partial evaluation. Just as a binding time analysis is used to compute the static/dynamic property, 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 online strategy that performs everything at once.

Our approach overcomes the limitations (*i*), (*ii*) and (*iii*) mentioned above. Let us summarize the new contributions of this paper.

- 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.

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 online parameterized partial evaluation. In particular, Section 4.1 presents the notion of facet together with examples and Section 4.4 describes the

1. Syntactic Domains

$c \in \mathbf{Const}$ Constants
$x \in \mathbf{Var}$ Variables
$p \in \mathbf{Po}$ Primitive Operators
$f \in \mathbf{Fn}$ Function Names
$e \in \mathbf{Exp}$ Expressions

$e ::= c \mid x \mid p\ (e_1, \cdots, e_n) \mid f\ (e_1, \cdots, e_n) \mid if\ e_1\ e_2\ e_3$

$\mathbf{Prog} ::= \{f_i(x_1, \cdots, x_n) = e_i\}$ (f_1 is the main function)

2. Semantics Domains

$b \in \mathbf{Values} = (\mathbf{Int} + \mathbf{Bool})_\perp$
$\rho \in \mathbf{Env} = \mathbf{Var} \rightarrow \mathbf{Values}$
$\Theta \in \mathbf{FunEnv} = \mathbf{Fn} \rightarrow \mathbf{Values}^n \rightarrow \mathbf{Values}$

3. Valuation Functions

$\mathcal{E}_{Prog} : \mathbf{Prog} \rightarrow \mathbf{Values}$
$\mathcal{E} : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{FunEnv} \rightarrow \mathbf{Values}$
$\mathcal{K} : \mathbf{Const} \rightarrow \mathbf{Values}$
$\mathcal{K}_P : \mathbf{Po} \rightarrow \mathbf{Values}^n \rightarrow \mathbf{Values}$

$$\mathcal{E}_{Prog}\ [\![\{ f_i(x_1, \cdots, x_n) = e_i\}]\!] =$$
$$\Theta\ [\![f_1]\!]\ \ where rec\ \Theta = \perp[(\lambda(b_1, \cdots, b_n)\ .\ \mathcal{E}\ [\![e_i]\!]\ (\perp[b_k/x_k])\ \Theta)/f_i]$$
$$\mathcal{E}\ [\![c]\!]\ \rho\ \Theta = \mathcal{K}\ [\![c]\!]$$
$$\mathcal{E}\ [\![x]\!]\ \rho\ \Theta = \rho\ [\![x]\!]$$
$$\mathcal{E}\ [\![p(e_1, \cdots, e_n)]\!]\ \rho\ \Theta = \mathcal{K}_P\ [\![p]\!]\ (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta, \cdots,\ \mathcal{E}\ [\![e_n]\!]\ \rho\ \Theta)$$
$$\mathcal{E}\ [\![if\ e_1\ e_2\ e_3]\!]\ \rho\ \Theta = (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta) \rightarrow \mathcal{E}\ [\![e_2]\!]\ \rho\ \Theta,\ \mathcal{E}\ [\![e_3]\!]\ \rho\ \Theta$$
$$\mathcal{E}\ [\![f(e_1, \cdots, e_n)]\!]\ \rho\ \Theta = \Theta\ [\![f]\!]\ (\mathcal{E}\ [\![e_1]\!]\ \rho\ \Theta, \cdots, \mathcal{E}\ [\![e_n]\!]\ \rho\ \Theta)$$

Figure 1: Standard Semantics of a First Order Language

semantics of online parameterized partial evaluation. Section 5 presents offline parameterized partial evaluation. We introduce the notion of abstract facet in Section 5.1 and present facet analysis in Section 5.4. In Section 5.5, we extend the facet analysis to handle higher order programs. Section 6 presents an example of online and offline parameterized partial evaluation. Section 7 concludes and discusses future work.

2 Preliminaries

In this section we examine conventional partial evaluation for strict functional programs. For conciseness we only consider first order programs; although, as discussed in Section 5.5, extending the framework to higher order programs is straightforward using existing techniques.

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 $\mathcal{K}$ maps a constant to its semantic value; function $\mathcal{K}_P$ defines the usual semantic operations for primitive operators. Domain **FunEnv** maps function names to their meaning. The result of a program is the value of f_1. We assume all functions 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 ([3, 21, 6], for example). The figure only highlights aspects of the semantics relevant to later discussion. For example, we omit details about treatment of function calls (unfolding and specialization). 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 representation of the corresponding specialized functions. Essentially, this achieves *instantiation* and *folding* as in [5], and ensures uniqueness of specialized functions. To keep track of each specialization, partial evaluation is single-threaded with respect to the cache. 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.

Because partial evaluation is a source-to-source program transformation, it operates on a syntactic domain — denoted by **Exp**. Domain **FnEnv** recursively defines the meaning of each function. Function $\mathcal{K}^{-1}$ maps a value (*e.g.*, integer and boolean) back to its textual representation.

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.

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

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

3. Valuation Functions

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

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

$$\mathcal{SPE}\,[\![c]\!]\,\rho\,\varpi\,\sigma = \langle [\![c]\!],\,\sigma\rangle$$

$$\mathcal{SPE}\,[\![x]\!]\,\rho\,\varpi\,\sigma = \langle \rho\,[\![x]\!],\,\sigma\rangle$$

$$\mathcal{SPE}\,[\![p(e_1,\cdots,e_n)]\!]\,\rho\,\varpi\,\sigma = \langle \mathcal{SK}_P\,[\![p]\!]\,(e'_1,\cdots,e'_n),\,\sigma_n\rangle \ \textit{where}\ \langle e'_1,\,\sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma,\ \vdots,\ \langle e'_n,\,\sigma_n\rangle = \mathcal{SPE}\,[\![e_n]\!]\,\rho\,\varpi\,\sigma_{n-1}$$

$$\mathcal{SPE}\,[\![if\ e_1\ e_2\ e_3]\!]\,\rho\,\varpi\,\sigma = (e'_1 \in \textbf{Const}) \rightarrow (\mathcal{K}\,e'_1) \rightarrow \mathcal{SPE}\,[\![e_2]\!]\,\rho\,\varpi\,\sigma_1,\ \mathcal{SPE}\,[\![e_3]\!]\,\rho\,\varpi\,\sigma_1,\ \langle [\![if\ e'_1\ e'_2\ e'_3]\!],\,\sigma_3\rangle \ \textit{where}\ \langle e'_2,\,\sigma_2\rangle = \mathcal{SPE}\,[\![e_2]\!]\,\rho\,\varpi\,\sigma_1,\ \langle e'_3,\,\sigma_3\rangle = \mathcal{SPE}\,[\![e_3]\!]\,\rho\,\varpi\,\sigma_2 \ \textit{where}\ \langle e'_1,\,\sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma$$

$$\mathcal{SPE}\,[\![f(e_1,\cdots,e_n)]\!]\,\rho\,\varpi\,\sigma = APP\,[\![f]\!]\,e'_1\cdots e'_n\,\sigma_n\,\varpi \ \textit{where}\ \langle e'_1,\,\sigma_1\rangle = \mathcal{SPE}\,[\![e_1]\!]\,\rho\,\varpi\,\sigma,\ \vdots,\ \langle e'_n,\,\sigma_n\rangle = \mathcal{SPE}\,[\![e_n]\!]\,\rho\,\varpi\,\sigma_{n-1}$$

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

Figure 2: Simple Partial Evaluation Semantics

3 The Abstraction Methodology

This section presents a general methodology to introduce abstract values in the partial evaluation process. Sections 4 and 5 describe how to instantiate this methodology for online and offline partial evaluation.

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 particular, a concrete algebra can be captured by the notion of *semantic algebra* as defined in denotational semantics (*e.g.*, [19]).

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

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 [1, 16], this can be formally achieved by relating the two algebras with an abstraction function. Because we aim at addressing both online and offline partial evaluation, a given algebra may be defined at three different levels — listed in increasing abstractness: standard semantics, online partial evaluation and offline 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.

Notationally, a symbol s is noted $\hat{s}$ if it is used in online partial evaluation and $\tilde{s}$ in offline partial evaluation. Symbols that refer to standard semantics are unannotated. Finally, for generality, any symbol used in either online or

offline partial evaluation is noted $\overline{s}$.

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 [D; 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 $\widehat{\mathbf{D}} = \{\bot, pos, zero, neg, \top\}$.

Domains $\widehat{\mathbf{D}}$ and $\mathbf{D}$ are related by the following abstraction function.

$$\begin{array}{lcll} \widehat{\alpha}_{\widehat{D}} & : & \mathbf{D} \rightarrow \widehat{\mathbf{D}} & \\ \widehat{\alpha}_{\widehat{D}}(x) & = & \bot_{\widehat{D}} & if\ d = \bot_D \\ & & pos & if\ d > 0 \\ & & zero & if\ d = 0 \\ & & neg & if\ d < 0 \end{array}$$

This example is further developed in Section 4.1.

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 [A; O], we say that $p \in \mathbf{O}$ is closed if and only if $p : \mathbf{A} \rightarrow \mathbf{A}$. Thus, the abstract version of a closed operator will be passed abstract values to compute new ones; this corresponds to an abstract primitive in abstract interpretation.

The second class of operators consists of those whose co-domain 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 [A; 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 co-domain. Since an operator may be defined at three different levels (standard semantics, online and offline partial evaluation), its corresponding co-domain 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 online 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 offline 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 co-domains. To do so let us define the abstraction functions relating the three levels of definition of domain Values.

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

$$\begin{array}{lcll} \widehat{\tau} & : & \mathbf{Values} \rightarrow \widehat{\mathbf{Values}} & \\ \widehat{\tau}(x) & = & \bot_{\widehat{Values}} & if\ x = \bot_{Values} \\ & & \mathcal{K}^{-1}\ x & otherwise \end{array}$$

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

Note that to be consistent with our framework, domain $\widehat{\mathbf{Values}}$ used above denotes a separated sum constructed by adding the elements $\bot_{\widehat{Values}}$ and $\top_{\widehat{Values}}$ to the original domain of constants Const; these 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 $\mathcal{K}$); this domain is further discussed in Section 4.

To investigate the relation between online partial evaluation and offline partial evaluation, recall that conventional offline 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 $\bot_{\widetilde{Values}}$. This domain forms a chain, with ordering $\bot_{\widetilde{Values}} \sqsubseteq Static \sqsubseteq Dynamic$, and abstracts the online partial evaluation process in the following way.

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

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 [A; O] and [A'; O'] be two algebras, and $\alpha_{A'} : \mathbf{A} \rightarrow \mathbf{A}'$ and $\overline{\tau} : \mathbf{B} \rightarrow \overline{\mathbf{Values}}$ be two abstraction functions. Then, the algebra abstraction is noted $\alpha_{A'} : [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A}'; \mathbf{O}']$.

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

1. $\mathbf{A}'$ *is an algebraic lattice of finite height*[1].

[1] An algebraic lattice is defined as an algebraic CPO that is also a complete lattice. All lattices defined in this paper are algebraic lattices. 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]).

2. *If $p \in \mathbf{O}$ is a closed operator, then $p' : \mathbf{A}' \to \mathbf{A}'$ is its corresponding abstract version.*

3. *If $p \in \mathbf{O}$ is an open operator with functionality $\mathbf{A} \to \mathbf{B}$, where $\mathbf{B}$ is some domain different from $\mathbf{A}$, then $p' : \mathbf{A}' \to \overline{\mathbf{Values}}$ is its corresponding abstract version.*

4. *$\forall p \in \mathbf{O}$ and its corresponding abstract version $p' \in \mathbf{O}'$*
$\alpha_{A'} \circ p \sqsubseteq p' \circ \alpha_{A'}$ *if p is a closed operator*
$\overline{\tau} \circ p \sqsubseteq p' \circ \alpha_{A'}$ *if p is an open operator with functionality $\mathbf{A} \to \mathbf{B}$*

Notice that Condition 1 ensures termination in computing abstract values. Also, for simplicity, we only consider a limited form of heterogeneous algebra (Conditions 2 and 3): only the co-domain of an operator can be different from the carrier of the algebra. Finally, Condition 4 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* [18, 16].

Definition 3 (Logical Relation $\sqsubseteq_{\alpha_{A'}}$) *Let $\alpha_{A'} : [\mathbf{A};\mathbf{O}] \to [\mathbf{A}';\mathbf{O}']$ be a facet mapping with respect to $\overline{\mathbf{Values}}$. We define the binary 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 $p : \mathbf{A} \to \mathbf{B}$ for some domain $\mathbf{B}$. Then*
$p \sqsubseteq_{\alpha_{A'}} p' \Leftrightarrow$
$\forall a \in \mathbf{A}, \forall a' \in \mathbf{A}' : a \sqsubseteq_{\alpha_{A'}} a' \Rightarrow p(a) \sqsubseteq_{\overline{\tau}} p'(a')$

where $\sqsubseteq_{\overline{\tau}}$ is the logical relation defined for the facet mapping $\overline{\tau} : [\mathbf{B};\mathbf{O}_B] \to [\overline{\mathbf{Values}};\mathbf{O}'_B]$. Facet mappings $\hat{\tau}$ and $\tilde{\tau}$ are presented in Definitions 7 and 10 respectively.

Using this logical relation, we can re-formulate the safety criteria expressed by Condition 4 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 $\overline{\mathbf{Values}}$, $\forall p \in \mathbf{O}$ and its corresponding abstract version $p' \in \mathbf{O}'$, $p \sqsubseteq_{\alpha_{A'}} p'$.*

4 Online 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. Then, we 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 $[\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ defined by a facet mapping $\widehat{\alpha}_{\widehat{D}} : [\mathbf{D};\mathbf{O}] \to [\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ with respect to $\widehat{\mathbf{Values}}$.*

We refer to $\widehat{\mathbf{D}}$ as the facet domain and $\widehat{\mathbf{O}}$ 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, we have $\forall \hat{d}_i \in \widehat{\mathbf{D}}$ and $\forall d_i \in \mathbf{D}$ such that $d_i \sqsubseteq_{\widehat{\alpha}_{\widehat{D}}} \hat{d}_i$ for $i \in \{1,\ldots,n\}$:*

$$\hat{p}(\hat{d}_1,\cdots,\hat{d}_n) \in \mathbf{Const} \text{ and } p(d_1,\cdots,d_n) \text{ terminates}$$
$$\Rightarrow \hat{p}(\hat{d}_1,\cdots,\hat{d}_n) = \widehat{\tau}\,(p(d_1,\cdots,d_n))$$

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 $\sqsubseteq_{\widehat{\alpha}_{\widehat{D}}}$.

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 $\{\bot, pos, zero, neg, \top\}$. 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 $[\mathbf{D};\mathbf{O}] = [\mathbf{Int}_\bot; \{+, <\}]$.

1. $\widehat{\mathbf{D}} = \{\bot, pos, zero, neg, \top\}$ with
$\forall \hat{d} \in \widehat{\mathbf{D}}. \bot \sqsubseteq \hat{d} \sqsubseteq \top$
2. The abstraction function is
$$\widehat{\alpha}_{\widehat{D}} : \mathbf{D} \to \widehat{\mathbf{D}}$$
$$\widehat{\alpha}_{\widehat{D}}(x) = \begin{cases} \bot_{\widehat{D}} & \text{if } d = \bot_D \\ pos & \text{if } d > 0 \\ zero & \text{if } d = 0 \\ neg & \text{if } d < 0 \end{cases}$$
3. $\widehat{\mathbf{O}} = \widehat{O}_o \cup \widehat{O}_c$ where $\widehat{O}_o = \{\hat{<}\}$ and $\widehat{O}_c = \{\hat{+}\}$
4. Facet operators
$$\hat{+} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \to \widehat{\mathbf{D}}$$
$$\hat{+} = \lambda\,(\hat{d}_1, \hat{d}_2).\ (\hat{d}_1 = \bot) \vee (\hat{d}_2 = \bot) \to \bot,\ \hat{d}_1 = zero \to \hat{d}_2,\ \hat{d}_2 = zero \to \hat{d}_1, \hat{d}_1 \sqcup \hat{d}_2$$

$$\hat{<} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \rightarrow \widehat{\mathbf{Values}}$$
$$\hat{<} = \lambda\ (\hat{d}_1,\ \hat{d}_2).$$
$$(\hat{d}_1 = \bot) \vee (\hat{d}_2 = \bot) \rightarrow \bot_{\widehat{Values}},$$
$$(\hat{d}_1 = pos) \wedge (\hat{d}_2 \in \{neg,\ zero\}) \rightarrow [false],$$
$$(\hat{d}_1 = zero) \wedge (\hat{d}_2 = pos) \rightarrow [true],$$
$$(\hat{d}_1 = zero) \wedge (\hat{d}_2 \in \{neg,\ zero\}) \rightarrow [false],$$
$$(\hat{d}_1 = neg) \wedge (\hat{d}_2 \in \{pos,\ zero\}) \rightarrow [true],$$
$$\top_{\widehat{Values}}$$

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 useful results of any facet computation 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 $\widehat{\alpha}_i : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{D}}_i;\widehat{\mathbf{O}}_i]$ 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 $[\widehat{\mathcal{D}};\widehat{\Omega}]$, consists of two components:*

1. *A domain $\widehat{\mathcal{D}} = \widehat{\mathbf{D}}_1 \otimes \cdots \otimes \widehat{\mathbf{D}}_m \cong \prod_{i=1}^{m} \widehat{\mathbf{D}}_i$; it is a smashed product [2] of the facet domains;*
2. *A set of product operators $\widehat{\Omega}$ such that for any $p \in \mathbf{O}$ and its corresponding product operator $\hat{\omega}_p \in \widehat{\Omega}$,*

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

   (b) otherwise, $p \in \mathbf{O}$ is an open operator
   $p : \mathbf{D}^n \rightarrow \mathbf{D}'$ *for some domain $\mathbf{D}'$, and*
   $\hat{\omega}_p : \widehat{\mathcal{D}}^n \rightarrow \widehat{\mathbf{Values}}$
   $$\hat{\omega}_p = \lambda(\hat{\delta}_1,\cdots,\hat{\delta}_n)\ .$$
   $$(\exists j \in \{1,\cdots,m\}\ s.t.\ \hat{\delta}^j = \bot_{\widehat{Values}}) \rightarrow \bot_{\widehat{Values}},$$
   $$(\exists j \in \{1,\cdots,m\}\ s.t.\ \hat{\delta}^j \in \mathbf{Const}) \rightarrow \hat{\delta}^j, \top_{\widehat{Values}}$$
   $$where\ \ \hat{\delta} = \prod_{i=1}^{m} \hat{p}_i(\hat{\delta}_1^i,\cdots,\hat{\delta}_n^i)$$

[2] Given two lattices $\mathbf{D}$ and $\mathbf{E}$, its smashed product, $\mathbf{D} \otimes \mathbf{E}$, is a lattice, the elements of which are defined by the function, *smashed*, such that:

$$smashed : \mathbf{D} \times \mathbf{E} \rightarrow \mathbf{D} \otimes \mathbf{E}$$
$$smashed(d,e) = \begin{cases} (d,e) & if\ (d \neq \bot_D)\ and\ (e \neq \bot_E) \\ \bot_{D\otimes E} & otherwise \end{cases}$$

Notice that the i-th component of an element of the domain $\widehat{\mathcal{D}}$ is noted $\hat{\delta}^i$; this domain is partially ordered componentwise. All product operators defined above are monotonic.

Although facets of a product are defined independently, the facet values with respect to which a program is specialized must have some consistency.

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

$$\bigcap_{i=1}^{m} \{d \in \mathbf{D} \mid d \sqsubseteq_{\widehat{\alpha}_i} \hat{\delta}^i\}\ \textit{is not the empty set nor}\ \{\bot\}.$$

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

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 are more than one facet that produce concrete values, those values are equal (see proof in [8]).

Lemma 3 *Let $[\widehat{\mathcal{D}};\widehat{\Omega}]$ be a product of facets and $p \in \mathbf{O}$ be an open operator,*

$$If\ \ \exists\ j,k \in \{1,\cdots,m\}\ (j \neq k)\ and\ \hat{\delta}_1,\cdots,\hat{\delta}_n \in \widehat{\mathcal{D}}$$
$$such\ that\ both\ \hat{p}_j(\hat{\delta}_1^j,\cdots,\hat{\delta}_n^j) \in \mathbf{Const}\ and\ \hat{p}_k(\hat{\delta}_1^k,\cdots,\hat{\delta}_n^k) \in$$
$$\mathbf{Const},\ then\ \ \hat{p}_j(\hat{\delta}_1^j,\cdots,\hat{\delta}_n^j) = \hat{p}_k(\hat{\delta}_1^k,\cdots,\hat{\delta}_n^k)$$

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 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. Application of the same notion to partial evaluation raises the following question: What can be captured by a partial evaluation facet?

Just as a facet defines symbolic behavior of primitives, the partial evaluation facet will capture the partial evaluation behavior of primitives. 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* $\widehat{\alpha}_{\widehat{Values}} : [\mathbf{D};\mathbf{O}] \to [\widehat{\mathbf{Values}};\widehat{\mathbf{O}}]$

1. $\widehat{\alpha}_{\widehat{Values}} : \mathbf{D} \to \widehat{\mathbf{Values}}$
   $\widehat{\alpha}_{\widehat{Values}} \equiv \widehat{\tau}$

2. $\forall\, \hat{p} \in \widehat{\mathbf{O}}$ *of arity* n

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

In fact, the abstraction function $\widehat{\alpha}_{\widehat{Values}}$ is essentially the same as $\widehat{\tau}$ given in Section 3.2: it maps a value into its textual representation.

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

We can now define the semantics of parameterized partial evaluation.

4.4 Semantics of Online Parameterized Partial Evaluation

Since this semantics aims at defining partial evaluation, we shall 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 $\widehat{\mathcal{SD}}$; each summand corresponds to a semantic algebra. We shall use $\hat{\delta}$ to denote an element of domain $\widehat{\mathcal{SD}}$. For readability, we do not index $\hat{\delta}$ with a given summand and assume that it denotes an element of the appropriate summand. As before, $\hat{\delta}'$ denotes i-th facet value a product of facet values.

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.

For a product of facets $\widehat{\mathcal{D}}$, $\hat{\alpha}_{\widehat{D}_i}$ denotes the i-th abstraction function. Besides computing facet values, the partial evaluator has to construct the residual program and collect the specialized functions. This triple forms the codomain of the partial evaluation function and is defined as $\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf}$. 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 $\widehat{\mathcal{K}}$ and $\widehat{\mathcal{K}}_P$ propagate this value to all facets in a product by invoking their corresponding abstraction function.

The following theorem asserts that any constant produced by partial evaluating a primitive call is always correct with respect to the standard semantics, modulo termination (see proof in [8]).

Theorem 1 *Let* $[\widehat{\mathcal{D}};\widehat{\Omega}]$ *be a product of facets (including the partial evaluation facet) for an algebra* $[\mathbf{D};\mathbf{O}]$,
$\exists\, j \in \{1, \cdots, m\}$ *such that,*

$(c \in \mathbf{Const})$ *and* $(\mathcal{E}[\![p(x_1, \cdots, x_n)]\!]\bot[d_i/x_i]\bot)$ *terminates* $\Rightarrow\ \ c = \widehat{\tau}(\mathcal{E}[\![p(x_1, \cdots, x_n)]\!]\bot[d_i/x_i]\bot)$

where $c = (\mathcal{PE}[\![p(x_1, \cdots, x_n)]\!]\bot[\langle[\![x_i]\!], \hat{\delta}_i\rangle/x_i]\bot\bot)\downarrow 1$

and $\ \ d_i \in \{d \in \mathbf{D} \mid d \sqsubseteq_{\widehat{\alpha}_{\widehat{\mathcal{D}}_j}} \hat{\delta}_i'\}.$

Finally, let us point out that online partial evaluation as defined in Figure 3 provides a less complete treatment of conditional expressions than the one described in Redfun [13]. 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 Offline Parameterized Partial Evaluation

As discussed earlier, in an online 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 *offline* 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 offline parameterized partial evaluation, we generalize the binding time analysis to *facet analysis*: a

1. Semantics Domains

$$\widehat{\delta} \in \widehat{\mathcal{SD}} = \sum_{j=1}^{s} \widehat{\mathcal{D}}_j \quad \textit{where } \widehat{\mathcal{D}}_j = (\widehat{\mathbf{D}}_{j1} \otimes \cdots \otimes \widehat{\mathbf{D}}_{jm}) \textit{ and } s \textit{ is the number of basic domains}$$

$$e' \in \mathbf{Exp}$$

$$\rho \in \mathbf{Env} = \mathbf{Var} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}})$$

$$\varpi \in \mathbf{FnEnv} = \mathbf{Fn} \rightarrow \mathbf{Exp}^n \rightarrow \widehat{\mathcal{SD}}^n \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf})$$

$$\sigma \in \mathbf{Sf} = (\mathbf{Fn} \times \mathbf{Exp}^n \times \widehat{\mathcal{SD}}^n) \rightarrow \mathbf{Exp}$$

2. Valuation Functions

$$\mathcal{PE}_{Prog} : \mathbf{Prog} \rightarrow \mathbf{Exp}^n \rightarrow \widehat{\mathcal{SD}}^n \rightarrow \mathbf{Prog}_{\perp}$$

$$\mathcal{PE} : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{FnEnv} \rightarrow \mathbf{Sf} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf})_{\perp}$$

$$\widehat{\mathcal{K}}_P : \mathbf{Po} \rightarrow \mathbf{Exp}^n \rightarrow \widehat{\mathcal{SD}}^n \rightarrow \mathbf{Sf} \rightarrow (\mathbf{Exp} \times \widehat{\mathcal{SD}} \times \mathbf{Sf})_{\perp}$$

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

$$\mathcal{PE}[\![c]\!]\ \rho\ \varpi\ \sigma = \widehat{\mathcal{K}}[\![c]\!]\ \sigma$$

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

$$\mathcal{PE}[\![p(e_1,\cdots,e_n)]\!]\ \rho\ \varpi\ \sigma = \widehat{\mathcal{K}}_P[\![p]\!]\ \langle e'_1,\cdots,e'_n\rangle\ \langle \widehat{\delta}_1,\cdots,\widehat{\delta}_n\rangle\ \sigma_n$$
$$\textit{where}\quad \langle e'_1,\widehat{\delta}_1,\sigma_1\rangle = \mathcal{PE}[\![e_1]\!]\ \rho\ \varpi\ \sigma$$
$$\vdots \qquad\qquad \vdots$$
$$\langle e'_n,\widehat{\delta}_n,\sigma_n\rangle = \mathcal{PE}[\![e_n]\!]\ \rho\ \varpi\ \sigma_{n-1}$$

$$\mathcal{PE}[\![if\ e_1\ e_2\ e_3]\!]\ \rho\ \varpi\ \sigma = (e'_1 \in \mathbf{Const}) \rightarrow (\mathcal{K}\ e'_1) \rightarrow \mathcal{PE}[\![e_2]\!]\ \rho\ \varpi\ \sigma_1,\ \mathcal{PE}[\![e_3]\!]\ \rho\ \varpi\ \sigma_1,$$
$$\langle [\![if\ e'_1\ e'_2\ e'_3]\!], \widehat{\delta}, \sigma_3\rangle$$
$$\textit{where}\quad \langle e'_2,\widehat{\delta}_2,\sigma_2\rangle = \mathcal{PE}[\![e_2]\!]\ \rho\ \varpi\ \sigma_1$$
$$\langle e'_3,\widehat{\delta}_3,\sigma_3\rangle = \mathcal{PE}[\![e_3]\!]\ \rho\ \varpi\ \sigma_2$$
$$\widehat{\delta} = \widehat{\delta}_2 \sqcup \widehat{\delta}_3$$
$$\textit{where}\ \langle e'_1,\widehat{\delta}_1,\sigma_1\rangle = \mathcal{PE}[\![e_1]\!]\ \rho\ \varpi\ \sigma$$

$$\mathcal{PE}[\![f(e_1,\cdots,e_n)]\!]\ \rho\ \varpi\ \sigma = APP[\![f]\!]\ \langle e'_1,\cdots,e'_n\rangle\ \langle \widehat{\delta}_1,\cdots,\widehat{\delta}_n\rangle\ \sigma_n\ \varpi$$
$$\textit{where}\quad \langle e'_1,\widehat{\delta}_1,\sigma_1\rangle = \mathcal{PE}[\![e_1]\!]\ \rho\ \varpi\ \sigma$$
$$\vdots \qquad\qquad \vdots$$
$$\langle e'_n,\widehat{\delta}_n,\sigma_n\rangle = \mathcal{PE}[\![e_n]\!]\ \rho\ \varpi\ \sigma_{n-1}$$

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

$$\widehat{\mathcal{K}}_P[\![p^c]\!]\ \langle e'_1,\cdots,e'_n\rangle\ \langle \widehat{\delta}_1,\cdots,\widehat{\delta}_n\rangle\ \sigma = (\widehat{\delta} = \perp_{\widehat{\mathcal{D}}}) \rightarrow \langle [\![p^c(e'_1,\cdots,e'_n)]\!], \perp_{\widehat{\mathcal{SD}}}, \sigma\rangle$$
$$(\widehat{\delta}^1 \in \mathbf{Const}) \rightarrow \langle \widehat{\delta}^1, \langle \widehat{\alpha}_{\widehat{\mathcal{D}}_1}(d),\cdots,\ \widehat{\alpha}_{\widehat{\mathcal{D}}_m}(d)\rangle, \sigma\rangle,$$
$$\langle [\![p^c(e'_1,\cdots,e'_n)]\!], \widehat{\delta}, \sigma\rangle$$
$$\textit{where}\ p^c : \mathbf{D}^n \rightarrow \mathbf{D}$$
$$\widehat{\delta} = \widehat{\omega}_{p^c}(\widehat{\delta}_1,\cdots,\widehat{\delta}_n)$$
$$d = \mathcal{K}\ \widehat{\delta}^1$$

$$\widehat{\mathcal{K}}_P[\![p^o]\!]\ \langle e'_1,\cdots,e'_n\rangle\ \langle \widehat{\delta}_1,\cdots,\widehat{\delta}_n\rangle\ \sigma = (\hat{d} = \perp_{\widehat{Values}}) \rightarrow \langle e', \perp_{\widehat{\mathcal{SD}}}, \sigma\rangle,$$
$$\hat{d} \in \mathbf{Const} \rightarrow \langle \hat{d}, \langle \widehat{\alpha}_{\widehat{\mathcal{D}}'_1}(\hat{d}),\cdots,\widehat{\alpha}_{\widehat{\mathcal{D}}'_m}(\hat{d})\rangle, \sigma\rangle$$
$$\langle e', \langle \top_{\widehat{\mathcal{D}}'_1},\cdots,\ \top_{\widehat{\mathcal{D}}'_m}\rangle, \sigma\rangle$$
$$\textit{where}\ p^o : \mathbf{D}^n \rightarrow \mathbf{D}'$$
$$\hat{d} = \widehat{\omega}_{p^o}(\widehat{\delta}_1,\cdots,\widehat{\delta}_n)$$
$$d = \mathcal{K}\ \hat{d}$$
$$e' = [\![p^o(e'_1,\cdots,e'_n)]\!]$$

Figure 3: Online Parameterized Partial Evaluation

phase that statically computes facet information. Consequently, the task of program specialization reduces to following the information yielded by the facet analysis.

To present offline parameterized partial evaluation, we follow the approach used in defining online 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 lastly, 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* $[\widetilde{\mathbf{D}};\widetilde{\mathbf{O}}]$ *of a facet* $[\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ *is defined by a facet mapping* $\widetilde{\alpha}_{\widetilde{D}} : [\widehat{\mathbf{D}};\widehat{\mathbf{O}}] \to [\widetilde{\mathbf{D}};\widetilde{\mathbf{O}}]$ *with respect to* $\widetilde{\mathbf{Values}}$.

This definition leads to the following property about open operators.

Property 4 *For any open operator* $\hat{p} \in \widehat{\mathbf{O}}$ *of arity n, we have* $\forall \tilde{d}_1, \cdots, \tilde{d}_n \in \widetilde{\mathbf{D}}$ *and* $\forall \hat{d}_i$ *such that* $\hat{d}_i \sqsubseteq_{\widetilde{\alpha}_{\widetilde{D}}} \tilde{d}_i$ *for* $i \in \{1, \cdots, n\}$

$$(\tilde{p}(\tilde{d}_1, \cdots, \tilde{d}_n) = Static) \Rightarrow \hat{p}(\hat{d}_1, \cdots, \hat{d}_n) \sqsubseteq_{\widehat{Values}} c$$

for $c \in \mathbf{Const}$.

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 $[\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ is defined as follows.

1. $\widetilde{\mathbf{D}} = \widehat{\mathbf{D}}$ *(similar to Example 1)*

2. $\widetilde{\alpha}_{\widetilde{D}}$ is simply the identity mapping between $\widehat{\mathbf{D}}$ and $\widetilde{\mathbf{D}}$.

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

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

5.2 Product of Abstract Facets

As in online parameterized partial evaluation, we now define the *product of abstract facets.*

Definition 9 (Product of Abstract Facets) *Let* $\widetilde{\alpha}_i : [\widehat{\mathbf{D}}_i;\widehat{\mathbf{O}}_i] \to [\widetilde{\mathbf{D}}_i;\widetilde{\mathbf{O}}_i]$ *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* $[\widetilde{\mathcal{D}},\widetilde{\Omega}]$, *consists of two components:*

1. *A domain* $\widetilde{\mathcal{D}} = \prod_{i=1}^{m} \widetilde{\mathbf{D}}_i$ *is a smashed product of the abstract facet domains;*

2. *A set of product operators* $\widetilde{\Omega}$ *such that for any* $p \in \mathbf{O}$ *and its corresponding product operator* $\tilde{\omega}_p \in \widetilde{\Omega}$,

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

   (b) *otherwise,* $\tilde{p} \in \widetilde{\mathbf{O}}$ *is an open operator, and*
   $p : \mathbf{D}^n \to \mathbf{D}'$ *for some domain* $\mathbf{D}'$, *and*
   $\tilde{\omega}_p : \widetilde{\mathcal{D}}^n \to \widetilde{\mathbf{Values}}$
   $$\begin{array}{l} \tilde{\omega}_p = \lambda(\tilde{\delta}_1, \cdots, \tilde{\delta}_n)\ . \\ \qquad (\exists j \in \{1,\cdots,m\}\ s.t.\ \tilde{\delta}^j = \bot_{\widetilde{Values}}) \to \bot_{\widetilde{Values}}, \\ \qquad (\exists j \in \{1,\cdots,m\}\ s.t.\ \tilde{\delta}^j = Static) \to Static, \\ \qquad Dynamic \\ \qquad where\ \tilde{\delta} = \prod_{i=1}^{m} \tilde{p}_i(\tilde{\delta}_1^i, \cdots, \tilde{\delta}_n^i) \end{array}$$

The domain $\widetilde{\mathcal{D}}$ is partially ordered component-wise. Since all the product components are of finite height by definition, the product domain is also of finite height.

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* $[\widehat{\mathbf{Values}}; \widehat{\mathbf{O}}]$ *is defined by the facet mapping* $\widetilde{\alpha}_{\widetilde{Values}} : [\widehat{\mathbf{Values}}; \widehat{\mathbf{O}}] \rightarrow [\widetilde{\mathbf{Values}}; \widetilde{\mathbf{O}}]$

1. $\widetilde{\alpha}_{\widetilde{Values}} : \widehat{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}}$
   $\widetilde{\alpha}_{\widetilde{Values}} \equiv \widetilde{\tau}$

2. $\forall\, \tilde{o} \in \widetilde{\mathbf{O}}$ *of arity* n
   $\tilde{o} : \widetilde{\mathbf{Values}}^n \rightarrow \widetilde{\mathbf{Values}}$
   $\tilde{o} = \lambda\, (\tilde{d}_1, \cdots, \tilde{d}_n)\, .\, \bigwedge_{i=1}^{n} (\tilde{d}_i = \mathit{Static}) \rightarrow \mathit{Static}, \mathit{Dynamic}$

Not surprisingly, the above definition 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 [21] 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 $\widetilde{\mathcal{SD}}$, 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. The notational conventions about indices are similar to Figure 3. The analysis aims at collecting facet information for each function in a given program; this forms the *facet signature* of the function. More precisely, a facet signature consists of a product of abstract facet values for each parameter of a function and is defined as $\widetilde{\mathcal{SD}}^n$. 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 $\widetilde{\mathcal{E}}$ maps each user-defined function into its abstract version. The resulting abstract functions are then used by the valuation function $\widetilde{\mathcal{A}}$ to compute the facet signatures. As usual, computation is accomplished via fixpoint iteration. Functions $\widetilde{\mathcal{K}}$ and $\widetilde{\mathcal{K}}_P$ perform the abstract computation on constants and primitive operators. This is similar to functions $\widehat{\mathcal{K}}$ and $\widehat{\mathcal{K}}_P$ defined in Figure 3. Finally, note that fixpoint iteration is performed over the domains $\widetilde{\mathcal{SD}}$ 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.

5.5 Higher Order Offline Parameterized Partial Evaluation

The techniques for higher order online partial evaluation are now known (*e.g.*, [20, 12]). However, traditionally problems arise when dealing with the offline strategy. For this reason, this section will concentrate on offline parameterized partial evaluation. In particular, we will present a higher order facet analysis, the essential component of the offline strategy, just as binding time analysis for conventional offline partial evaluation.

While the first order facet analysis extends conventional first order binding time analysis, the higher order analysis makes use of recently developed technique in abstract interpretation for analyzing higher order programs (*e.g.*, [14, 15, 4, 7]).

With the introduction of higher order functions, the abstract version of each user-defined function may now take higher order abstract functions as arguments. This means that the abstract facet property should be captured by a domain consisting of both first-order and higher order properties:

$$\varphi \in \mathbf{Av} = \widetilde{\mathcal{SD}} + (\mathbf{Av} \rightarrow \mathbf{Av})$$

Figure 8 displays the facet analysis for higher order programs. The language has been extended to include higher order functions. Similar to the first-order facet analysis, function $\widetilde{\mathcal{E}}$ transforms a user-defined function into its abstract version, while function $\widetilde{\mathcal{A}}$ uses the abstract function to collect abstract facet information for each user-defined function. Given two functions f_1 and f_2, we define their meet as follows:

$$\begin{aligned} f_1 \sqcup f_2 = \; & (f_1 = \top_C) \vee (f_2 = \top_C) \rightarrow \top_C, \\ & (\mathit{arity}(f_1) = \mathit{arity}(f_2)) \rightarrow \\ & \quad \lambda(\varphi_1, \cdots, \varphi_n)\, .\, f_1(\varphi_1, \cdots, \varphi_n) \sqcup f_2(\varphi_1, \cdots, \varphi_n), \\ & \mathit{Serr} \end{aligned}$$

where $\top_C$ denotes an operator which always returns the appropriate strongest element in the domain **Av**; *Serr* denotes the error function.

$\widetilde{\mathcal{E}}$ performs fixpoint computation over the domain **Av** to produce the abstract version of the user-defined functions. The semantics is self-explanatory, except for the treatment of conditional expression: when a conditional expression returns a higher order function, the meet of the functions obtained from the two branches is returned. However, when the test expression is dynamic, the "unknown" operator $\top_C$ is returned to indicate that the possible operators returned by the conditional expression cannot be determined statically, and therefore will not be applied at specialization time. Functionally, $\top_C$ takes arbitrary number of arguments, and always returns the appropriate strongest elements in the domain **Av**. For convenience, we assume that it is pre-defined in the initial environment ϱ_0.

$\widetilde{\mathcal{A}}$ performs a global analysis to collect the facet signature of each user-defined function. These signatures are captured by the domain **SignEnv**. The co-domain of $\widetilde{\mathcal{A}}$ is defined as follows.

$$\mathbf{Ans} = \mathbf{SigEnv} \times ((\mathbf{Av}^n \times \mathbf{Ans}^n) \rightarrow \mathbf{Ans}) \times \mathcal{P}(\mathbf{Fn})$$

The first component consists of the facet signatures as described above. The other two components are only significant when the expression being analyzed is higher order. In this case, these components represent its abstract behavior in the following sense. When a higher order function is applied, it may induce new facet signatures; this is captured

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

$$\widetilde{\delta} \in \widetilde{\mathcal{SD}} = \sum_{j=1}^{s} \widetilde{\mathcal{D}}_j \text{ where } \widetilde{\mathcal{D}}_j = (\widetilde{\mathbf{D}}_{j1} \otimes \cdots \otimes \widetilde{\mathbf{D}}_{jm}) \text{ and } s \text{ is the number of basic domains}$$

$$\varrho \in \mathbf{Env} = \mathbf{Var} \to \widetilde{\mathcal{SD}}$$

$$\pi \in \mathbf{SigEnv} = \mathbf{Fn} \to \widetilde{\mathcal{SD}}^n$$

$$\varsigma \in \mathbf{FEnv} = \mathbf{Fn} \to \widetilde{\mathcal{SD}}^n \to \widetilde{\mathcal{SD}}$$

3. Valuation Functions

$$\widetilde{\mathcal{M}} : \mathbf{Program} \to \widetilde{\mathcal{SD}}^n \to \mathbf{SigEnv}$$

$$\widetilde{\mathcal{E}} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Fenv} \to \widetilde{\mathcal{SD}}$$

$$\widetilde{\mathcal{A}} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Fenv} \to \mathbf{SigEnv}$$

$$\widetilde{\mathcal{M}}\,[\![\{f_i(x_1,\cdots,x_n) = e_i\}]\!]\,\langle\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n\rangle = \widetilde{h}(\perp[\langle\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n\rangle/f_1])$$

$$\text{where rec } \widetilde{h}\,\pi = \pi \sqcup \widetilde{h}(\bigsqcup\{\widetilde{\mathcal{A}}\,[\![e_i]\!]\,(Ev\;\pi\;f_i)\;fenv \mid \forall f_i\})$$

$$Ev\;\pi\;f = \perp[(\pi\;f)\downarrow_k/x_k \mid \forall x_k \in \text{parameters of } f]$$

$$fenv = \perp[(\lambda(\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n)\,.\,\widetilde{\mathcal{E}}\,[\![e_i]\!]\,[\widetilde{\delta}_k/x_k]\;fenv)/f_i]$$

$$\widetilde{\mathcal{E}}\,[\![c]\!]\,\varrho\,\varsigma = \widetilde{\mathcal{K}}\,[\![c]\!]$$

$$\widetilde{\mathcal{E}}\,[\![x]\!]\,\varrho\,\varsigma = \varrho\,[\![x]\!]$$

$$\widetilde{\mathcal{E}}\,[\![p(e_1,\cdots,e_n)]\!]\,\varrho\,\varsigma = \widetilde{\mathcal{K}}_P\,[\![p]\!]\,(\widetilde{\mathcal{E}}\,[\![e_1]\!]\,\varrho\,\varsigma)\,\cdots\,(\widetilde{\mathcal{E}}\,[\![e_n]\!]\,\varrho\,\varsigma)$$

$$\widetilde{\mathcal{E}}\,[\![if\;e_1\;e_2\;e_3]\!]\,\varrho\,\varsigma = \widetilde{\delta}_1 = \perp_{\widetilde{\mathcal{SD}}} \to \perp_{\widetilde{\mathcal{SD}}},$$
$$\widetilde{\delta}_1^1 = Static \to \widetilde{\delta}_2 \sqcup \widetilde{\delta}_3,\ \langle Dynamic, \widetilde{\delta}_2^2 \sqcup \widetilde{\delta}_3^2, \cdots, \widetilde{\delta}_2^m \sqcup \widetilde{\delta}_3^m\rangle$$
$$\text{where } \widetilde{\delta}_i = \widetilde{\mathcal{E}}\,[\![e_i]\!]\,\varrho\,\varsigma \quad \text{for } i = \{1,2,3\}$$

$$\widetilde{\mathcal{E}}\,[\![f(e_1,\cdots,e_n)]\!]\,\varrho\,\varsigma = \varsigma\,[\![f]\!]\,((\widetilde{\mathcal{E}}\,[\![e_1]\!]\,\varrho\,\varsigma),\cdots,(\widetilde{\mathcal{E}}\,[\![e_n]\!]\,\varrho\,\varsigma))$$

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

$$\widetilde{\mathcal{K}}_P\,[\![p^c]\!]\,\widetilde{\delta}_1\cdots\widetilde{\delta}_n = \widetilde{\omega}_{p^c}(\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n) \text{ where } p^c : \mathbf{D}^n \to \mathbf{D}$$

$$\widetilde{\mathcal{K}}_P\,[\![p^o]\!]\,\widetilde{\delta}_1\cdots\widetilde{\delta}_n = \tilde{d} = \perp_{\widetilde{Values}} \to \perp_{\widetilde{\mathcal{SD}}}, \langle\tilde{d}, \top_{\widetilde{D}'_2},\cdots,\top_{\widetilde{D}'_m}\rangle$$
$$\text{where } p^o : \mathbf{D}^n \to \mathbf{D}'$$
$$\tilde{d} = \widetilde{\omega}_{p^o}(\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n)$$

$$\widetilde{\mathcal{A}}\,[\![c]\!]\,\varrho\,\varsigma = \perp$$

$$\widetilde{\mathcal{A}}\,[\![x]\!]\,\varrho\,\varsigma = \perp$$

$$\widetilde{\mathcal{A}}\,[\![p(e_1,\cdots,e_n)]\!]\,\varrho\,\varsigma = \bigsqcup_{i=1}^{n} \widetilde{\mathcal{A}}\,[\![e_i]\!]\,\varrho\,\varsigma$$

$$\widetilde{\mathcal{A}}\,[\![if\;e_1\;e_2\;e_3]\!]\,\varrho\,\varsigma = \bigsqcup_{i=1}^{3} \widetilde{\mathcal{A}}\,[\![e_i]\!]\,\varrho\,\varsigma$$

$$\widetilde{\mathcal{A}}\,[\![f(e_1,\cdots,e_n)]\!]\,\varrho\,\varsigma = (\bigsqcup_{i=1}^{n} \widetilde{\mathcal{A}}\,[\![e_i]\!]\,\varrho\,\varsigma) \sqcup \perp[\langle\widetilde{\delta}_1,\cdots,\widetilde{\delta}_n\rangle/f]$$
$$\text{where } \widetilde{\delta}_i = \widetilde{\mathcal{E}}\,[\![e_i]\!]\,\varrho\,\varsigma \quad \text{for } i = \{1,\cdots,n\}$$

Figure 4: Facet Analysis

by the second component of Ans. Also, when a user-defined function is applied, we would like to update its facet signature; this is captured by the third component of Ans, which consists of the set of possible user-defined functions that an expression may evaluate to.

Let us now explain the treatment of a conditional expression by function $\widetilde{\mathcal{A}}$. Besides the usual tasks of collecting signatures from each component of the conditional, we also need to determine if the abstract facet information returned is the unknown operator. Since this operator indicates that the higher order functions returned by both branches will not be applied at the specialization time, we must therefore apply the appropriate strongest abstract facet values to these functions in order to collect the signature information from their function bodies. The application is performed "in advance" before we return an answer from evaluating the conditional.

For simplicity, we assume that both the initial environments ϱ_0 and ς_0 contain functions that deal with primitive operations. Thus, for each primitive p, we have

$$\varrho_0\ [\![p]\!]\ \varphi_1 \cdots \varphi_n\ =\ \widetilde{\mathcal{K}}_P\ [\![p]\!]\ \varphi_1 \cdots\ \varphi_n$$

where $\widetilde{\mathcal{K}}_P$ is defined in Figure 4, and

$$\varsigma_0\ [\![p]\!](\langle\ \varphi_1 \cdots \varphi_n\rangle, \langle a_1, \cdots, a_n\rangle)\ =\ \langle \bot, Serr, \{\}\rangle.$$

In general, the analysis as described is not guaranteed to terminate. The reason is that a program may include higher order functions that need to be analyzed an infinite number of times. This situation is described by Hudak and Young in [14]; they circumvent the problem by disallowing functions whose type is of arbitrary "order" or "depth". Here, we adopt the same restriction.

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** → **V** creates a vector of the specified size.
UpdVec : **V** × **Int** × **Float** → **V** updates an element.
Vec‖ : **V** → **Int** returns the size of the vector.
Vref : **V** × **Int** → **Float** returns a specified element.

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 Online Parameterized Partial Evaluation

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

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

```
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)
```

Figure 5: Inner Product Program

2. Abstraction function

$$\widehat{\alpha_V} : \mathbf{V} \to \widehat{\mathbf{V}}$$
$$\widehat{\alpha_V}(v) = \begin{cases} \bot_{\widehat{V}} & \text{if } v = \bot \\ Vec\|(v) & \text{otherwise} \end{cases}$$

3. Closed operators

$$\widehat{MkVec} : \widehat{\mathbf{Values}} \to \widehat{\mathbf{V}}$$
$$\widehat{MkVec}(i) = (i = \bot_{\widehat{Values}}) \to \bot_{\widehat{V}},\ (i = \top_{\widehat{Values}}) \to \top_{\widehat{V}},\ i$$

$$\widehat{UpdVec} : \widehat{\mathbf{V}} \times \widehat{\mathbf{Values}} \times \widehat{\mathbf{Values}} \to \widehat{\mathbf{V}}$$
$$\widehat{UpdVec}(\hat{v}, i, r) = (i = \bot_{\widehat{Values}}) \vee (r = \bot_{\widehat{Values}}) \to \bot_{\widehat{V}},\ \hat{v}$$

4. Open operators

$$\widehat{Vec\|} : \widehat{\mathbf{V}} \to \widehat{\mathbf{Values}}$$
$$\widehat{Vec\|}(\hat{v}) = (\hat{v} = \bot_{\widehat{V}}) \to \bot_{\widehat{Values}},\ (\hat{v} = i) \to i,\ \top_{\widehat{Values}}$$

$$\widehat{Vref} : \widehat{\mathbf{V}} \times \widehat{\mathbf{Values}} \to \widehat{\mathbf{Values}}$$
$$\widehat{Vref}(\hat{v}, i) = (\hat{v} = \bot_{\widehat{V}}) \vee (i = \bot_{\widehat{Values}}) \to \bot_{\widehat{Values}},\ \top_{\widehat{Values}}$$

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

$$\texttt{<A, <}\top_{\widehat{Values}}\texttt{, 3>>}\ \text{and}\ \texttt{<B, <}\top_{\widehat{Values}}\texttt{, 3>>}$$

where A and B are residual identifiers for iprod; $\top_{\widehat{Values}}$ is the partial evaluation facet value; and, 3 is the size facet value. 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 non-recursive. Since elements of the vectors are unknown at partial evaluation time, the primitive operation Vref cannot be reduced; therefore, both the multiplication and addition operations are residual.

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

Figure 6: Residual Program for Inner Product

6.2 Offline Parameterized Partial Evaluation

In the offline parameterized partial evaluation, we define the abstract **Size** facet $[\widetilde{\mathbf{V}}; \widetilde{\mathbf{O}}]$.

1. $\widetilde{\mathbf{V}} = \{\perp_{\widetilde{V}}, s, d\}$ with the ordering $\perp_{\widetilde{V}} \sqsubseteq s \sqsubseteq d$.

   Values s and d denote a static and a dynamic vector size, respectively.

2. Abstraction function

$$\widetilde{\alpha_V} : \widehat{\mathbf{V}} \to \widetilde{\mathbf{V}}$$
$$\widetilde{\alpha_V}(\hat{v}) = \begin{cases} \perp_{\widetilde{V}} & \text{if } \hat{v} = \perp_{\widehat{V}} \\ d & \text{if } \hat{v} = \top_{\widehat{V}} \\ s & \text{otherwise} \end{cases}$$

3. Closed operators

$$\widetilde{MkVec} : \widetilde{\mathbf{Values}} \to \widetilde{\mathbf{V}}$$
$$\widetilde{MkVec}(x) = (i = \perp_{\widetilde{Values}}) \to \perp_{\widetilde{V}},\ (i = Dynamic) \to d,\ s$$

$$\widetilde{UpdVec} : \widetilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \times \widetilde{\mathbf{Values}} \to \widehat{\mathbf{V}}$$
$$\widetilde{UpdVec}(\tilde{v}, i, k) = (i = \perp_{\widetilde{Values}}) \vee (r = \perp_{\widetilde{Values}}) \to \perp_{\widetilde{V}}, \tilde{v}$$

4. Open operators

$$\widetilde{Vec\sharp} : \widetilde{\mathbf{V}} \to \widetilde{\mathbf{Values}}$$
$$\widetilde{Vec\sharp}(\tilde{v}) = (\tilde{v} = \perp_{\widetilde{V}}) \to \perp_{\widetilde{Values}},\ (\tilde{v} = s) \to Static, Dynamic$$

$$\widetilde{Vref} : \widetilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \to \widetilde{\mathbf{Values}}$$
$$\widetilde{Vref}(\tilde{v}, i) = (\tilde{v} = \perp_{\widetilde{V}}) \vee (i = \perp_{\widetilde{Values}}) \to \perp_{\widetilde{Values}},\ Dynamic$$

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 `iProd` will then be bound to the pair of abstract values `<Dynamic, S>`. 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 online 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 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 an abstract syntax tree rooted by the open operation `Vec♯`). Binding time analysis is the only facet computation performed for `dotProd`. This contrasts with the online parameterized partial evaluation of the inner product program where the size facet computations were performed for each function manipulating vectors.

7 Conclusion and Future Works

Redfun is the main approach aimed at specializing programs with respect to static properties. Since then other partial evaluation systems (*e.g.*, [20, 12, 2]) have been developed based on this approach.

Parameterized partial evaluation goes beyond this in that: it captures both online and offline partial evaluation; the notion of facet provides a formal method to introduce user-defined static properties; finally facet analysis achieves efficiency of the specialization phase by enabling self-application.

Furthermore, our approach subsumes conventional self-applicable partial evaluation *à la* Mix [17] in that it generalizes the notion of binding time analysis to any static properties.

We are currently implementing parameterized partial evaluation for higher order functional programs and 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.

Acknowledgements

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] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, 1990.

[3] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[4] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1990.

[5] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

[6] C. Consel. *Analyse de Programmes, Evaluation Partielle et Generation de Compilateurs.* PhD thesis, Université de Paris VI, Paris, France, 1989.

[7] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[8] C. Consel and S. C. Khoo. Parameterized partial evaluation. Research report, Yale University, New Haven, Connecticut, USA, 1991. Extended version.

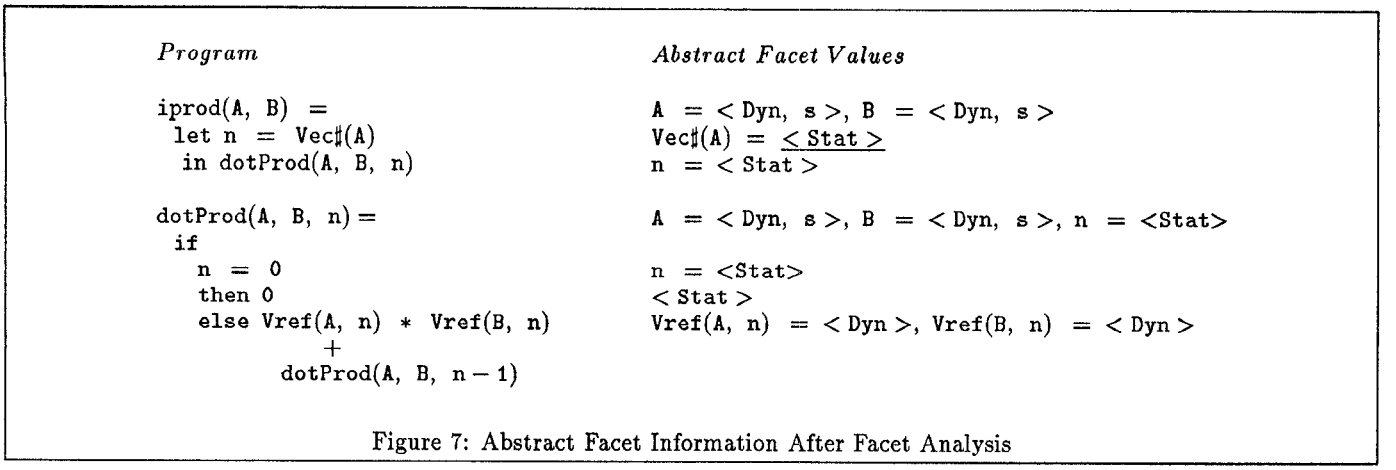

Figure 7: Abstract Facet Information After Facet Analysis

[9] P. Cousot and R. Cousot. 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*, pages 238–252, 1977.

[10] P. Emanuelson and A. Haraldsson. On compiling embedded languages in Lisp. In *ACM Conference on Lisp and Functional Programming, Stanford, California*, pages 208–215, 1980.

[11] H. Ganzinger and N. D. Jones, editors. *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[12] M. A. Guzowski. Toward developing a reflexive partial evaluator for an interesting subset of Lisp. Master's thesis, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, 1988.

[13] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linkoping University, Sweden, 1977. Linkoping Studies in Science and Technology Dissertations N° 14.

[14] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[15] P. Hudak and J. Young. A collecting interpretation of expressions (without Powerdomains). In *ACM Symposium on Principles of Programming Languages*, pages 107–118, 1988.

[16] N. D Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. Technical report, University of Copenhagen and Aarhus University, Copenhagen, Denmark, 1990.

[17] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[18] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[19] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[20] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, M.I.T. (LCS), Massachusetts, U.S.A, 1984.

[21] P. Sestoft. The structure of a self-applicable partial evaluator. In [11], pages 236–256, 1985.

1. Syntactic Domains

$e \in \mathbf{Exp}$ Expressions

$f \in \mathbf{Fn}$ Functions

$e ::= c \mid x \mid f \mid if\ e_1\ e_2\ e_3 \mid \lambda(x_1, \cdots, x_n)\ e \mid e\ (e_1, \cdots, e_n)$

2. Semantics Domains

$\varphi \in \mathbf{Av} = \widetilde{\mathcal{SD}} + \mathbf{Av} \to \mathbf{Av}$

$\pi \in \mathbf{SigEnv} = \mathbf{Fn} \to \mathbf{Av}^n$

$a \in \mathbf{Ans} = \mathbf{SigEnv} \times ((\mathbf{Av}^n \times \mathbf{Ans}^n) \to \mathbf{Ans}) \times \mathcal{P}(\mathbf{Fn})$

$\varrho \in \mathbf{Env} = \mathbf{Var} \to \mathbf{Av}$

$\varsigma \in \widetilde{\mathbf{Env}} = \mathbf{Var} \to (\mathbf{Av}^n \times \mathbf{Ans}^n) \to \mathbf{Ans}$

3. Valuation Functions

$\widetilde{\mathcal{M}} : \mathbf{Program} \to \widetilde{\mathcal{SD}}^n \to \mathbf{SigEnv}$

$\widetilde{\mathcal{E}} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Av}$

$\widetilde{\mathcal{A}} : \mathbf{Exp} \to \mathbf{Env} \to \widetilde{\mathbf{Env}} \to \mathbf{Ans}$

$\widetilde{\mathcal{M}}\ [\![\{f_i(x_1, \cdots, x_n) = e_i\}]\!]\ \langle \varphi_1, \cdots, \varphi_n \rangle =$
$(\varsigma\ [\![f_1]\!]\ \langle \varphi_1, \cdots, \varphi_n \rangle\ \langle \bot, \cdots, \bot \rangle) \downarrow 1$
$where rec\ \varrho = \varrho_0[(\lambda\ (\varphi_1, \cdots, \varphi_n)\ .\ \widetilde{\mathcal{E}}\ [\![e_i]\!]\ \varrho[\varphi_k/x_k])/f_i]$
$\varsigma = \varsigma_0[(\lambda\ (\langle \varphi_1, \cdots, \varphi_n \rangle, \langle a_1, \cdots, a_n \rangle)\ .\ \widetilde{\mathcal{A}}\ [\![e_i]\!]\ \varrho[\varphi_k/x_k]\ \varsigma[a_k/x_k])/f_i]$

$\widetilde{\mathcal{E}}\ [\![c]\!]\ \varrho = \langle \widetilde{\Gamma}_1(d), \cdots, \widetilde{\Gamma}_m(d) \rangle\ where\ \widetilde{\Gamma}_i = \widetilde{\alpha}_{\widetilde{\mathcal{D}}_i} \circ \widehat{\alpha}_{\widehat{\mathcal{D}}_i}\ and\ d = (\mathcal{K}\ c)$

$\widetilde{\mathcal{E}}\ [\![x]\!]\ \varrho = \varrho\ [\![x]\!]$

$\widetilde{\mathcal{E}}\ [\![f]\!]\ \varrho = \varrho\ [\![f]\!]$

$\widetilde{\mathcal{E}}\ [\![if\ e_1\ e_2\ e_3]\!]\ \varrho = \varphi_1 = \bot_{Av} \to \bot_{Av},$
$(\varphi_1^1 = Static) \to \varphi_2 \sqcup \varphi_3,$
$\varphi_2 \in \widetilde{\mathcal{SD}} \to \langle Dynamic, \varphi_2^2 \sqcup \varphi_3^2, \cdots, \varphi_2^m \sqcup \varphi_3^m \rangle,\ \top_C$
$where\ \varphi_i = \widetilde{\mathcal{E}}\ [\![e_i]\!]\ \varrho\ \varsigma\ \ for\ i = \{1, 2, 3\}$

$\widetilde{\mathcal{E}}\ [\![\lambda(x_1, \cdots, x_n)\ e]\!]\ \varrho = \lambda(\varphi_1, \cdots, \varphi_n)\ .\ \widetilde{\mathcal{E}}\ [\![e]\!]\ (\varrho[\varphi_k/x_k])$

$\widetilde{\mathcal{E}}\ [\![e(e_1, \cdots, e_n)]\!]\ \varrho = (\widetilde{\mathcal{E}}\ [\![e]\!]\ \varrho)\ (\widetilde{\mathcal{E}}\ [\![e_1]\!]\ \varrho, \cdots, \widetilde{\mathcal{E}}\ [\![e_n]\!]\ \varrho)$

$\widetilde{\mathcal{A}}\ [\![c]\!]\ \varrho\ \varsigma = \langle \bot, Serr, \{\} \rangle$

$\widetilde{\mathcal{A}}\ [\![f]\!]\ \varrho\ \varsigma = \langle \bot, \varsigma\ [\![f]\!], \{f\} \rangle$

$\widetilde{\mathcal{A}}\ [\![x]\!]\ \varrho\ \varsigma = \varsigma\ [\![x]\!]$

$\widetilde{\mathcal{A}}\ [\![if\ e_1\ e_2\ e_3]\!]\ \varrho\ \varsigma = \varphi = \top_C \to \langle \pi \sqcup \pi', \top_F, \{\} \rangle,\ \langle \pi, g, F_2 \sqcup F_3 \rangle$
$where\ \langle \pi', -, - \rangle = g\ (\langle \top, \cdots, \top \rangle, \langle \top, \cdots, \top \rangle)$
$\langle \pi_i, g_i, F_i \rangle = \widetilde{\mathcal{A}}\ [\![e_i]\!]\ \varrho\ \varsigma\ \ for\ i = \{1, 2, 3\}$
$\varphi = \widetilde{\mathcal{E}}\ [\![if\ e_1\ e_2\ e_3]\!]\ \varrho$
$\pi = \pi_1 \sqcup \pi_2 \sqcup \pi_3$
$g = g_2 \sqcup g_3$

$\widetilde{\mathcal{A}}\ [\![\lambda(x_1, \cdots, x_n)\ e]\!]\ \varrho\ \varsigma = \langle \bot, \lambda(\langle \varphi_1, \cdots, \varphi_n \rangle, \langle a_1, \cdots, a_n \rangle)\ .\ \widetilde{\mathcal{A}}\ [\![e]\!]\ \varrho[\varphi_k/x_k]\ \varsigma[a_k/x_k], \{\} \rangle$

$\widetilde{\mathcal{A}}\ [\![e(e_1, \cdots, e_n)]\!]\ \varrho\ \varsigma = \langle \pi' \sqcup \pi'', g', F' \rangle$
$where\ \langle \pi', g', F' \rangle = g\ (\langle \varphi_1, \cdots, \varphi_n \rangle, \langle a_1, \cdots, a_n \rangle)$
$\pi'' = \pi[\langle \varphi_1, \cdots, \varphi_n \rangle / f \mid \forall f \in F] \sqcup \bigsqcup_{i=1}^{n} \pi_i$
$\langle \pi, g, F \rangle = \widetilde{\mathcal{A}}\ [\![e]\!]\ \varrho\ \varsigma$
$\varphi_i = \widetilde{\mathcal{E}}\ [\![e_i]\!]\ \varrho\ \ for\ i = \{1, \cdots, n\}$
$a_i = \langle \pi_i, -, - \rangle = \widetilde{\mathcal{A}}\ [\![e_i]\!]\ \varrho\ \varsigma\ \ for\ i = \{1, \cdots, n\}$

$\top_F\ (\langle \varphi_1, \cdots, \varphi_n \rangle, \langle a_1, \cdots, a_n \rangle) = \langle \bot, \top_F, \{\} \rangle$

Figure 8: Facet Analysis for higher order Programs