Parameterized Partial Evaluation
Principle and Practice

Siau Cheng Khoo
Research Report YALEU/DCS/RR-926
June 1992

This work is supported by the National University of Singapore and NSF
CCR-8809919

Parameterized Partial Evaluation
Principle and Practice

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Siau Cheng Khoo
November 1992

Abstract

Parameterized Partial Evaluation

Principle and Practice

Siau Cheng Khoo

Yale University

1992

Partial evaluation aims at specializing a program with respect to part of the input that is known. This process yields a new program which is a specialized version of the original program. This specialized program is expected to be more efficient than the original one.

In practice, there are two apparently independent strategies of partial evaluation: on-line and off-line. An on-line partial evaluator processes a program in one single phase, whereas an off-line partial evaluator performs some analyses before specializing the program.

Regardless of strategies used, most existing partial evaluators have the limitation that they only specialize program with respect to actual values. Specializing programs with respect to *static* properties about the input (such as signs, ranges, and types) is a natural extension of current partial evaluation and significantly contributes towards adapting partial evaluation to a larger variety of applications. Although work has been done in this direction, there has not been a formal treatment of this idea, and the systems developed thus far do not provide users with the capability of introducing static properties into the partial-evaluation process.

This thesis introduces the notion of *parameterized partial evaluation* – a generic form of partial evaluation parameterized with respect to user-defined static properties. This generality is accomplished by introducing an algebraic framework that enables modular definition of static properties and systematic incorporation of these properties into the partial-evaluation process. Consequently, new kinds of partial-evaluation applications become possible through the introduction of various static properties.

Not only does the framework guarantee the safety of the partial-evaluation process with respect to the static properties introduced, but it also defines a formal relationship between on-line and off-line partial evaluation. Moreover, it enables us to prove the correctness of partial evaluation with polyvariant specialization (in which any function in a program can have more than one specialized version), which has not been done before.

Finally, the effectiveness of parameterized partial evaluation is demonstrated through an implementation for a first-order strict functional language with data structures. Some applications are experimented to show the qualitative improvement of the residual programs produced using the parameterized partial evaluator.

Parameterized Partial Evaluation
Principle and Practice

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Siau Cheng Khoo
November 1992

Acknowledgments

I would like to thank my advisors, Paul Hudak and Charles Consel. Paul introduced me to the field of partial evaluation. He has provided me with encouragement, inspiration, and the freedom to do what I wanted to do. Charles strengthened my ability to thrive in this dynamic field of research. His thoughtful insights, enormous enthusiasm and perseverance, and careful guidance have brought my research to fruition. The numerous long discussions and sometimes bitter arguments that I had with Charles have not only shaped my research, but also influenced my life positively. Both Paul and Charles brought me into academia by teaching me about writing papers and giving talks, and doing independent research. To them, I express my deepest gratitude.

I would also like to thank Olivier Danvy for his timely advice and encouragement over the years. Not only has he provided valuable and meticulous criticism about my work, but he has always never forgotten to boost up my research spirit by crediting those little accomplishments I have made along the way.

Many thanks go to Young-il Choo, for expounding domain theory and category theory, for his careful reading of this dissertation, and for his helpful comments.

National University of Singapore has generously provided my financial support and study leave necessary to pursue this work. I am indebted to Professor Yuen Chung Kwong at National University of Singapore for advising me on how to do research when I was in deep depression in the second year of my PhD program.

My gratitude also goes to the members of the Yale Haskell Group, who provided heartfelt friendship and useful advice in numerous wrestling sessions, thesis-defense practices, and lunches. Special thank goes to Amir Kishon, who was always around to listen to my grumbling and provide creative (and sometimes wild) suggestions; to Dan Rabin, who corrected many grammatical errors in this document; to Tom Blenko, who helped improve my defense presentation; to Kung Chen, who kept me updated of issues outside partial evaluation; and to Linda Joyce, who provided efficient administrative support.

Without the company of many people my stay in the United States would not have been amiable and inspiring. I offer my thanks to everyone who has assisted me in whatever way during the last five years, especially to Soo Teck Lee, for contributing much laughter throughout the last three years; and to Hong-Hung Tam, for initiating many thought-provoking conversations.

Finally, I would like to thank my family: my wife Lian Sang, for her understanding, encouragement, and patience; my mother, for her confidence in me and her countless prayers; and my brothers and sisters-in-law, for their constant supports. It is to this heartwarming family I dedicate this dissertation.

Contents

List of Tables

List of Figures

Parameterized Partial Evaluation
Principle and Practice

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Siau Cheng Khoo

November 1992

Chapter 1

Introduction

1.1 Partial Evaluation

Partial evaluation is the process of specializing a program with respect to part of the input that is known. It yields a new *residual* program which is a specialized version of the original program. A faithful partial evaluator must satisfy the following criterion:

> Suppose that $P(x, y)$ is a program with two arguments, whose first argument x is known (*i.e.*, *static*), but whose second argument y is unknown (*i.e.*, *dynamic*). Specialization of $P(x, y)$ yields a residual program $P_x(y)$ such that:
>
> $$\forall y,\ P(x, y) \ = \ P_x(y)$$
>
> provided the evaluation of both $P(x, y)$ and $P_x(y)$ terminates.

In theory, partial evaluation can be viewed as a realization of the S_n^m theorem in recursive function theory [Kleene, 1952]. In practice, a partial evaluator is a source-to-source program transformation, and is expected to produce more efficient programs [Jones, 1990]. The partial-evaluation phase can be seen as a staging of the computations of a program: expressions that only operate on known data (called the *static expressions*) are executed during this phase; for the others (called the *dynamic expressions*), residual expressions are generated. This staging improves the execution time (called the *run-time*) of the residual program compared to the original program.

Two strategies of partial evaluation can be identified in practice. There are called *on-line* and *off-line* partial evaluation respectively.

1.1.1 On-Line Partial Evaluation

An on-line partial evaluator processes a program in one single phase (*e.g.,* [Haraldsson, 1977,Weise and Ruf, 1990]). It uses information available in the current context to make a decision about how to partially evaluate an expression. Thus, given a conditional expression

```
if b then 3+5 else 4*7
```

,

if `b` is found to be static in some context, partially evaluating the conditional in that context may yield the residual expression `8`. If, under another context, `b` is found to be dynamic, then both branches of the conditional will be partially evaluated, the conditional construct will be made residual, and the residual expression produced is: `if b then 8 else 28`.

1.1.2 Off-Line Partial Evaluation

In contrast to on-line partial evaluation, an off-line partial evaluator (*e.g.,* [Jones *et al.*, 1989,Bondorf, 1990,Consel, 1990b]) divides the task of partial evaluation into two phases: an *analysis* phase followed by a *specialization* phase.

The main analysis performed is *binding-time analysis* (therefore, off-line partial evaluation is also called binding-time-based partial evaluation.) It determines which expressions within a program can be evaluated by the partial evaluator, given a known/unknown division of the program input. Using binding-time analysis, we can annotate each expression in the program as either static or dynamic. The static expressions are evaluated at partial-evaluation time (*i.e.*, specialization time), and the dynamic expressions are evaluated at run time.

Determining exact binding-time information is undecidable, so in practice, binding-time analysis is performed by an *abstract interpretation* which approximates the de-

sired property. Consider again the conditional expression given in the previous section. Even though `b` may be known in one context and unknown in another, binding-time analysis may conclude that `b` is dynamic as a safe approximation in all contexts. Hence, the result of partially evaluating this conditional expression in all contexts will be: `if b then 8 else 28`.

In the specialization phase, the *specializer* partially evaluates the annotated program (produced in the analysis phase) and the partially-known input. It behaves like an on-line partial evaluator, except that it uses the annotation of each expression to determine how to partially evaluate that expression.

Intuitively, binding-time analysis can naturally be viewed as an abstraction of the on-line partial-evaluation process. Unfortunately, prior to this thesis, this has not been proven, nor even been stated formally. Viewing binding-time analysis as an abstraction of on-line partial evaluation not only provides a more intuitive relationship between these two partial-evaluation strategies, but also enables the transfer of techniques developed for one strategy to another.

1.1.3 Applications

As a program transformation tool, partial evaluation has been applied to a wide variety of problems. Listing all existing applications of partial evaluation is beyond the scope of this thesis, but we can highlight some common applications, as well as some new ones. Reader can refer to [Sestoft, 1990] for a list of other applications.

1. **Pattern matching.** Partial evaluation has been used to generate a more dedicated pattern matcher by specializing a general pattern matcher with respect to a specific pattern [Emanuelson, 1980,Bondorf, 1988,Consel and Danvy, 1989,Jørgensen, 1990,Danvy, 1991]. For example, in [Consel and Danvy, 1989], a naive string matching algorithm is partially evaluated with respect to a pattern, producing a residual program that is essentially the Knuth-Morris-Pratt algorithm.

2. **Compilation.** The most publicized application for partial evaluation is the specialization of an interpreter with respect to a program. The residual program produced is a more efficient version of the original program, with much of the interpretive overhead removed. This essentially achieves the effect of compilation. Numerous experiments have been done in applying this technique to interpreters of different languages [Emanuelson and Haraldsson, 1980,Jones *et al.*, 1985,Safra, 1990,Consel and Khoo, 1991b]. Further optimizing compilation can be achieved using this technique by specializing the partial evaluator itself with respect to an interpreter, and by specializing the partial evaluator with respect to itself [Jones *et al.*, 1985,Jones *et al.*, 1989]. The former produces a optimizing compiler, whereas the latter produces a optimizing compiler generator. These three different levels of usage of partial evaluation are called the *Futamura projections* [Futamura, 1971].

3. **Incremental compilation.** Recently, Lombardi's work on incremental computation [Lombardi and Raphael, 1964,Lombardi, 1967] has been taken up by Sundaresh to build a formal framework for incremental computation based on partial evaluation [Sundaresh and Hudak, 1991,Sundaresh, 1991]. In particular, a formal methodology is introduced which uses partial evaluation as a tool to build incremental programs. The incrementality is obtained by maintaining a cache of residual functions (result of partially evaluating a function). Based on the algebraic properties of the residual functions, Sundaresh identifies the "combining operator" that combines two residual functions in such a way that re-computation in either one of the residual functions can be avoided.

4. **Program execution monitoring.** Partial-evaluation technique is recently used to produce practical program execution monitors in [Kishon, 1992,Kishon *et al.*, 1991]. Based on continuation semantics, Kishon provides a formal model for specifying the behavior of a large family of program execution monitors. Partial evaluation is then called upon to specialize an interpreter with respect

to a monitor specification; this yields an instrumented interpreter which automatically combines the standard interpretation and monitoring activities. Furthermore, specializing this instrumented interpreter with respect to a source program yields an instrumented program in which the extra code to perform monitoring has been automatically embedded.

1.2 Specializing Programs Using Static Properties

Besides specializing programs with respect to constant values, it is often necessary to specialize programs with respect to static properties such as signs, ranges, and types. Doing so 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 [Haraldsson, 1977] and carried out in practice with a system called Redfun in the late seventies.

Redfun partially evaluates Interlisp programs. It manipulates symbolic values such as data types to describe the possible values of a variable and a processed expression. As an example, consider the expression (in Interlisp)

```
(COND ((FIXP x) (IPLUS x x)) (T x))
```

Suppose that `x` is of type integer, then the expression can be reduced to `(IPLUS x x)`, even though the actual value of `x` is unknown during partial evaluation. Furthermore, from the operation `IPLUS`, Redfun deduces that the residual expression `(IPLUS x x)` is again of type integer, and this type information can further be used in partially evaluating the expression in which `(IPLUS x x)` is embedded.

Although the work on Redfun certainly started in the right direction, it has some limitations:

1. The static properties used in the systems are fixed, and thus cannot be supplied by the user. Since the kind of static properties to be used at partial-evaluation

time usually depends on the kind of application at hand, having a fixed set of static properties limits the system's versatility.

2. The use of static properties is not formally defined: There is no safety condition for the definition of symbolic values, no finiteness criteria for fixed-point iteration, *etc.* This hinders the opportunity for generalizing the approach to a larger class of static properties.

3. It is computationally expensive, primarily for two reasons:

   (a) It is an *on-line* partial evaluator. Since the treatment of an expression within a program is determined as it gets processed, it consists of numerous symbolic values and program transformations. (As a by-product, Redfun could not be self-applied as noticed in [Beckman *et al.*, 1976,Haraldsson, 1977,Emanuelson and Haraldsson, 1980], and thus, the partial-evaluation process could not be improved.)

   (b) It contains a fixed set of static properties. Computations on this set of static properties are always performed, even though some of these properties do not contribute any useful information during partial evaluation of a particular program.

Redfun is not alone in having these limitations. In fact, since the mid-eighties, other partial evaluation systems have been developed with similar capabilities and limitations (*e.g.*, [Schooler, 1984,Guzowski, 1988,Berlin, 1990,Weise and Ruf, 1990]).

An ideal solution to this problem is a partial evaluator that accepts user-defined static properties and utilizes them in a safe manner, without compromising its modularity. This means that such a partial evaluator should be *parameterized* with respect to the static properties. Functionally, in addition to the usual input (*i.e.*, a program and its input), such a partial evaluator should also accept a set of static properties.

1.3 Parameterized Partial Evaluation

The work described in this thesis grew out of a perceived need for a more general and formal treatment of partial evaluation using static properties, with the hope of developing a formal framework for partial evaluation such that static properties can be introduced safely and uniformly into both on-line and off-line strategies.

The thesis describes a generic form of partial evaluation, called *parameterized partial evaluation.* It differs from conventional partial evaluation in that the partial evaluation process is now parameterized with respect to static properties about program and its input. This enables the flexible encapsulation of properties of interest for any given application.

Since the partial-evaluation process has been parameterized, *dedicated partial-evaluation process* can be obtained by providing different kinds of user-defined static properties; each of these dedicated partial evaluators is at least as powerful as conventional partial evaluators. Consequently, new applications for partial evaluation become possible.

We introduce an *algebraic abstraction methodology* to model user-defined static properties. This methodology defines the various components constituting a static property, and imposes safety criteria for its use. Not only does the methodology capture the definition of static properties used in partial evaluation, but it also captures the partial-evaluation behavior of the primitive operations used.

Use of this abstraction methodology is not limited to on-line parameterized partial evaluation. Indeed, the same methodology is used to introduce static properties in the analysis phase of off-line parameterized partial evaluation. (The properties used in the specialization phase are similar to those used by on-line parameterized partial evaluation.) The analysis used by off-line parameterized partial evaluation is called a *facet analysis.* It allows pre-computation of static properties before actual partial evaluation, keeping the specialization phase simple. This is a natural extension of binding-time analysis used in conventional partial evaluation. Lastly, analogous to

the on-line counterpart, the abstraction methodology can capture the binding-time behavior of the primitives. Thus, it provides a *uniform* approach to introducing (and utilizing) static properties in both on-line and off-line partial evaluation.

Based on the abstraction methodology, we describe a *formal framework* that defines the parameterized partial-evaluation process. This framework is both *sound* and *general*: it enables the correctness properties of partial evaluation to be stated and proved, and provides a *uniform* approach to the formalization of both on-line and off-line parameterized partial evaluation. The latter enables the definition of a *formal relationship* between on-line and off-line partial evaluation.

Finally, the effectiveness of parameterized partial evaluation is demonstrated through a practical implementation. Since static properties are introduced into partial evaluation only when they are required for an application, we can readily show that parameterized partial evaluation has an edge over the existing work in this area of research in that it avoids computation over properties which are not used in the application.

The work described in this thesis is done for the partial evaluation of a first-order applicative language. Extension to the higher order case is discussed in Chapter 5.

1.4 Thesis Organization

The remainder of this chapter lists the notation used in the rest of the thesis. Chapter 2 describes the modeling of static properties and the method for parameterizing the partial-evaluation process. Chapter 3 provides the formal semantic specifications and correctness properties (and proofs) of parameterized partial evaluation. Chapter 4 discusses the implementation and some applications of parameterized partial evaluation. Finally, in Chapter 5, we conclude the thesis and put the work into perspective.

1.5 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" [Jones and Nielson, 1990]. 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 $\mathbf{D}_1 \times \mathbf{D}_2$ denote 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:

$$
\begin{array}{lcl}
\mathit{smashed} & : & \mathbf{D}_1 \times \mathbf{D}_2 \to \mathbf{D}_1 \otimes \mathbf{D}_2 \\
\mathit{smashed}(d,e) & = & \langle d_1, d_2 \rangle \quad \mathit{if}\ (d_1 \neq \perp_{D_1})\ \mathit{and}\ (d_2 \neq \perp_{D_2}) \\
 & & \perp_{D_1 \otimes D_2} \quad \mathit{otherwise}
\end{array}
$$

All domain/sub-domain coercions are omitted when clear from context.

The ordering on functions $f,\ f' \in \mathbf{D}_1 \to \mathbf{D}_2$ is defined in the standard way: $f \sqsubseteq f' \Leftrightarrow (\forall d \in \mathbf{D}_1)\ f(d) \sqsubseteq f'(d)$. A function $f \in \mathbf{D}_1 \to \mathbf{D}_2$ is *monotonic* if 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 $f(\bigsqcup\{d_i\}) = \bigsqcup\{f(d_i)\}$ for any chain $\{d_i\} \subseteq \mathbf{D}_1$. A function $f \in \mathbf{D}_1 \to \mathbf{D}_2$ is said to be *strict* if $f(\bot_{D_1}) = \bot_{D_2}$. It is *⊥-reflecting* [Abramsky, 1990] if $f\ a = \bot_{D_2} \Rightarrow a = \bot_{D_1}$. An element $d \in \mathbf{D}$ is a *fixed point* of $f \in \mathbf{D} \to \mathbf{D}$ iff $f(d) = d$; it is the *least fixed point* if for every other fixed point d', we have that $d \sqsubseteq d'$. The composition of function $f \in \mathbf{D}_1 \to \mathbf{D}_2$ with $f' \in \mathbf{D}_2 \to \mathbf{D}_3$ is denoted by $f' \circ f$. We write $Dom(f)$ to denote the domain of f, and $Ran(f)$ to denote the range of 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 i-th 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 i-th 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: λv . *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 form 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 thesis describes three levels of evaluation: 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 $\overline{s}$.

Finally, an algebra is noted $[\mathbf{A}; \mathbf{O}]$ where $\mathbf{A}$ is the *carrier* of the algebra and $\mathbf{O}$ a set of functions operating on this domain.

Chapter 2

Modeling Static Properties

We begin our discussion of parameterized partial evaluation (abbreviated as PPE) by investigating how static properties about program input can be modeled such that:

1. they can be used during partial evaluation in a safe manner, and

2. they can be introduced uniformly into both on-line and off-line PPE.

In this chapter, we develop an algebraic abstraction methodology, called *facet mapping*, 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. It is to be used by on-line PPE.

The safety criteria of this abstraction are also captured by the facet mapping. The mapping is defined using abstract interpretation [Abramsky and Hankin, 1987,Jones and Nielson, 1990]. 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; *i.e.*, the primitive functions produce values that can be used during partial evaluation to reduce some expressions.

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.

Facet mapping as an abstraction methodology is general enough to model static properties used in the analysis phase of off-line PPE. (Since the static properties used in the off-line specialization phase are similar to that used in the on-line PPE phase, when we refer to the static properties used in off-line PPE, we always refer to those used in the analysis phase, unless stated otherwise.) These static properties are called *abstract facets*. Just as binding-time analysis is used to statically compute the static/dynamic property, we introduce a *facet analysis* to statically compute properties. Again, the uniform nature of facet-mapping definition enables us to capture the binding-time behavior of primitive functions as an abstract facet, whose domain contains the binding-time information and operations are primitive functions.

Once the modeling of static properties has been presented, we will describe how to parameterize the partial-evaluation process to achieve communication between a partial evaluator (or a facet analysis, for an off-line parameterized partial evaluator) and these properties.

To facilitate the comprehension of the modeling process, we first give a simple example in which use of PPE is desirable.

2.1 Inner-Product Example

Consider a program which computes the inner product of two 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.

$MkVec$: $\mathbf{Int} \rightarrow \mathbf{V}$	creates an empty vector of the specified size
$UpdVec$: $\mathbf{V} \times \mathbf{Int} \times \mathbf{Float} \rightarrow \mathbf{V}$	updates an element
$Vec\sharp$: $\mathbf{V} \rightarrow \mathbf{Int}$	returns the size of the vector
$Vref$: $\mathbf{V} \times \mathbf{Int} \rightarrow \mathbf{Float}$	returns a specified element of a given vector

```
fun iprod(A,B) =
  let n = Vec#(A)
   in dotProd(A,B,n)

fun dotProd(A,B,n) =
  if n <= 0 then 0
     else Vref(A,n) * Vref(B,n)
          + dotProd(A,B,n-1)
```

Figure 2.1: Program for Inner-Product Computation
`iprod` is the main function. It has two parameters of vector type, `A` and `B`. Evaluating `iprod` includes computing the size of `A` and invoking recursive function `dotProd` to perform the actual inner-product computation.

The program for computing inner product is presented in Figure 2.1. Notice that parameter `n` is the sole induction variable for the recursive function `dotProd`. Therefore, we may instruct the partial evaluator to unfold (*i.e.*, in-line) all recursive calls to `dotProd` when its argument `n` is known during partial evaluation. This yields a residual program with linear code. Suppose that we want to partially evaluate this program with respect to any pair of vectors of size 3, we would expect the residual program to look like that in Figure 2.2.

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

Figure 2.2: Desired Residual Program
Desired residual program produced by partially evaluating the inner-product program with respect to any pair of vectors of size 3.

The size information in this example represents a particular static property about the vector which we would like to capture and have it used by the partial evaluator. In general, we may want to introduce other kinds of static properties into the partial-evaluation process, depending on the problem at hand; conventional partial evaluation has difficulty dealing with this kind of information, since it specializes program with respect to concrete values, not static properties.[1]

[1]In this example, the size information can be captured by Mogensen's partial evaluator, because

2.2 The Abstraction Methodology

This section presents a general methodology to introduce abstract values into the partial-evaluation process. Sections 2.3 and 2.4 describe, respectively, how to instantiate this methodology for on-line and off-line partial evaluation, and provide examples for each instantiation. (In this thesis, the words *abstract* and *abstraction* are used in the sense of *approximate* and *approximation* respectively.)

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 prompts us to use an algebraic approach to model static properties, analogous to the notion of *semantic algebra* in denotational semantics (*e.g.*, [Schmidt, 1986]).

Definition 2.1 (Semantic Algebra) *A* semantic algebra, $[\mathbf{D}; \mathbf{O}]$, *consists of a semantic domain* $\mathbf{D}$, *and a set of operations* $\mathbf{O}$ *over* $\mathbf{D}$.

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

Our approach consists of deriving, from the semantic algebra, an *abstract algebra* composed of an abstract domain — capturing the properties of interest — and a set of abstract primitives operating on this domain. Using *abstract interpretation* [Abramsky and Hankin, 1987,Jones and Nielson, 1990], 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, which, listed in increasing abstractness, are the standard semantics, on-line partial evaluation, and off-line partial evaluation. Algebras used at these three levels of are called semantic (or concrete) algebras, facets, and abstract facets respectively.

---

it takes into consideration *partially-static data* [Mogensen, 1988], which contains information about the *structure* of the vector. However, using partially-static data to capture the size information defeats the purpose of defining the vector as an abstract data type. Furthermore, partially-static data can only captures structural information, but not other kinds of information, such as "positive vector", "non-zero vector", *etc.* Lastly, the encoding of partially-static data may be residual after partial evaluation.

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 2.2.1 and 2.2.2 respectively. Then, this is formalized in Section 2.2.3 where the notion of relating two algebras is precisely defined, together with safety criteria.

2.2.1 Relating Domains

Domains can be related using an *abstraction function* [Cousot and Cousot, 1977]. Such a function is strict and continuous; it maps values in an initial domain to those in an abstract domain.

Following the example in Section 2.1, we wish to introduce some symbolic computations on vector sizes abstracted from the vector algebra $[\mathbf{V}; \mathbf{O}]$. We define an abstraction of the vector domain that captures the vector-size property. Let the abstract domain be $\widehat{\mathbf{V}} = \{\bot, s_0, s_1, s_2, \ldots, \top\}$ with $\forall \hat{v} \in \widehat{\mathbf{V}} : \bot \sqsubseteq \hat{v} \sqsubseteq \top$. For all $i \in \{0, 1, 2, \ldots\}$, s_i represents any vector of size i. Domains $\widehat{\mathbf{V}}$ and $\mathbf{V}$ are related by the following abstraction function.

$$\begin{array}{lcl} \widehat{\alpha}_{\widehat{V}} & : & \mathbf{V} \rightarrow \widehat{\mathbf{V}} \\ \widehat{\alpha}_{\widehat{V}}(v) & = & (v = \bot) \rightarrow \bot_{\widehat{V}},\ s_{Vec\sharp(v)} \end{array}$$

Technically, to facilitate later proofs, abstraction functions are required to be $\bot$-*reflecting.*

Abstraction functions are also used to define relations between the main domains used at the three levels of evaluation. Standard evaluation is performed on basic values, such as integers and booleans. We use domain **Values** to denote the sum of these basic-value domains.

As a program transformation, on-line partial evaluation manipulates syntactic constructs. Therefore, instead of operating on the basic values in **Values**, it operates on their textual representations (*i.e.*, *constants*). Let **Const** denotes the set of all constants, we define a constant domain $\widehat{\mathbf{Values}}$ that contains all constants used during on-line partial evaluation.

Definition 2.2 (Constant Domain $\widehat{\textbf{Values}}$) *The* constant domain $\widehat{\textbf{Values}}$ *consists of all constants in the set* **Const**, *augmented with the least element* $\bot_{\widehat{Values}}$ *and the top element* $\top_{\widehat{Values}}$.

The top element $\top_{\widehat{Values}}$ denotes absence of static information during partial evaluation.

From standard semantics to on-line partial evaluation, we define an abstraction function $\hat{\tau}$ that maps basic values into their textual representations:

$$\begin{array}{lcl} \hat{\tau} & : & \textbf{Values} \rightarrow \widehat{\textbf{Values}} \text{ and} \\ \hat{\tau}(x) & = & (x = \bot_{Values}) \rightarrow \bot_{\widehat{Values}}, \mathcal{K}^{-1}(x) \end{array}$$

where function $\mathcal{K}^{-1}$ maps a basic value to its textual representation — a literal constant. (In defining a standard semantics, it is common to define a function $\mathcal{K}$ that maps a constant to its denotable value. Thus, the role of function $\mathcal{K}^{-1}$ can be considered as the inverse of $\mathcal{K}$.)

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

Recall from Chapter 1 that conventional off-line partial evaluation consists of two phases: binding-time analysis and specialization. Binding-time analysis operates on the binding-time domain $\widetilde{\textbf{Values}}$, which is defined as follows:

Definition 2.3 (Binding-time Domain $\widetilde{\textbf{Values}}$) *The* binding-time domain $\widetilde{\textbf{Values}}$ *is composed of the set* {*static*, *dynamic*} *lifted with a least element* $\bot_{\widetilde{Values}}$.[2] *It forms a chain with the ordering* $\bot_{\widetilde{Values}} \sqsubset static \sqsubset dynamic$.

The abstraction function between $\widehat{\textbf{Values}}$ and $\widetilde{\textbf{Values}}$ is defined as follows:

[2]Note that this three-point domain refines the usual two-point domain {*static*, *dynamic*} in that it allows the detection of functions that are never invoked, and simple cases of non-terminating computations. Without the value $\bot_{\widetilde{Values}}$, these cases would be considered *static*.

$$\begin{array}{lcl} \widetilde{\tau} & : & \widehat{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}} \ \textit{and} \\ \widetilde{\tau}(x) & = & (x = \perp_{\widehat{Values}}) \rightarrow \perp_{\widetilde{Values}}, \\ & & \quad (x \in \mathbf{Const}) \rightarrow \textit{static}, \ \textit{dynamic} \end{array}$$

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

2.2.2 Relating Operations

When 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: *closed* and *open* operators.

Closed operators are 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 its co-domain is the carrier $\mathbf{A}$. 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. In the vector-size example, *UpdVec* is considered a closed primitive. As such, its abstract version can be defined as follows:

$$\begin{array}{l} \widehat{UpdVec} \ : \ \widehat{\mathbf{V}} \times \widehat{\mathbf{Values}} \times \widehat{\mathbf{Values}} \rightarrow \widehat{\mathbf{V}} \\ \widehat{UpdVec}(\hat{v}, c, r) \ = \ (c = \perp_{\widehat{Values}}) \vee (r = \perp_{\widehat{Values}}) \ \rightarrow \perp_{\widehat{V}}, \ \hat{v} \end{array}$$

The use of domain $\widehat{\mathbf{Values}}$ will become clear when we discuss open operators. Here, we simply notice that domain $\widehat{\mathbf{V}}$ is used in the definition in place of $\mathbf{V}$, since it is the carrier of the abstract algebra. The definition simply complies with the rule that updating a vector does not change the size of the vector. Therefore, the size information remains intact.

Open operators have their co-domains different from the carrier of the algebra. Intuitively, we want the abstract version of an open operator use abstract values to trigger useful partial-evaluation computation. Interestingly, we can relate this

division of operations to optimizing compilation where, typically, a phase collects properties (like the job of closed operations) and another triggers optimizations using these properties (like the job of open operations).

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 because their co-domains are not the carrier.

Indeed, for an open operation to trigger useful partial-evaluation computation (or binding-time-analysis computation in the analysis phase), it must produce a value that can be used during partial evaluation to reduce some expressions (or, respectively, a value that can be used during binding-time analysis to deduce the staticity of some expressions). That is, the open operation must yield a value in the main domain on which the latter computation performs. Since an operator may be defined at three different levels (standard semantics, on-line and off-line partial evaluation), its corresponding co-domain will be the main domain used at the respective level: in the standard semantics, an operator belongs to a semantic algebra; both open and closed operators produce basic values in **Values**. In on-line partial evaluation, an operator belongs to a facet; when it is open it produces a constant in $\widehat{\textbf{Values}}$, provided it is called with appropriate values (see Section 2.3). In off-line partial evaluation, an operator belongs to an abstract facet; when it is open it mimics the facet operator and thereby produces a binding-time value in $\widetilde{\textbf{Values}}$ (see Section 2.4).

Thus, in order to relate an open operator to its abstract version, we also have to relate their co-domains. This is achieved by the abstraction function $\widehat{\tau}$ between **Values** and $\widehat{\textbf{Values}}$, and function $\widetilde{\tau}$ between $\widehat{\textbf{Values}}$ and $\widetilde{\textbf{Values}}$.

Returning to our vector-size example in which we represent a vector size by the symbol s_i, we notice that the primitive *Vec*$\sharp$ is open. Using domain $\widehat{\textbf{Values}}$ as its co-domain, $\widehat{Vec\sharp}$ returns the vector size by converting its symbolic value into a constant.

$$\widehat{Vec\sharp} : \widehat{\mathbf{V}} \rightarrow \widehat{\mathbf{Values}}$$
$$\widehat{Vec\sharp}(\hat{v}) = (\hat{v} = \perp_{\widehat{V}}) \rightarrow \perp_{\widehat{Values}},$$
$$(\hat{v} = s_i) \rightarrow \hat{\tau}(i),\ \top_{\widehat{Values}}$$

2.2.3 Relating Algebras

Given this preliminary discussion we can now formalize 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. We extend function $\alpha_{A'}$ to one that maps from algebra $[\mathbf{A}; \mathbf{O}]$ to algebra $[\mathbf{A'}; \mathbf{O'}]$:

Definition 2.4 (Facet Mapping) $\alpha_{A'} : [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A'}; \mathbf{O'}]$ *is a* facet mapping with respect to α' *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. $\forall p \in \mathbf{O}$, $\exists p' \in \mathbf{O'}$ *such that*
   $\alpha_{A'} \circ p \sqsubseteq p' \circ \alpha_{A'}$ *if* p *is a closed operator*
   $\alpha' \circ p \sqsubseteq p' \circ \alpha_{A'}$ *if* p *is an open operator*

Condition *1* defines domain $\mathbf{A'}$ as a complete lattice; this imposes a partial ordering among its elements (*i.e.*, the abstract values). For any two elements $a_1, a_2 \in \mathbf{A'}$, we say "a_2 is *coarser* than a_1" if $a_1 \sqcup a_2 = a_2$. Condition *1* also ensures termination in computing abstract values. Lastly, Condition *3* states the safety criteria for defining abstract operators.

Given a facet mapping, we can succinctly describe the relationship between the components of two algebras by a *logical relation* [Nielson, 1989,Jones and Nielson, 1990].

[3]For a lattice of infinite height, a *widening* operator must be used to find fixed points in a finite number of steps (see [Cousot and Cousot, 1977]).

Definition 2.5 (Logical Relation $\preceq_{\alpha_{A'}}$) *A facet mapping* $\alpha_{A'} : [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A}'; \mathbf{O}']$ *with respect to* $\alpha' = \{\alpha_{B'_i} : \mathbf{B}_i \rightarrow \mathbf{B}'_i\}$ *induces a* logical relation $\preceq_{\alpha_{A'}}$ *as follows.*

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

2. $\forall\, b \in \mathbf{B},\ \forall\, b' \in \mathbf{B}'$: $b \preceq_{\alpha'} b' \Leftrightarrow \alpha_{B'}(b) \sqsubseteq_{B'} b'$.

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

4. *Let* $p \in \mathbf{O}$ *and* $p' \in \mathbf{O}'$ *be open operators and* $p : \mathbf{A} \rightarrow \mathbf{B}_i$ *for some domain* $\mathbf{B}_i$. *Then,*
   $p \preceq_{\alpha_{A'}} p' \Leftrightarrow \forall\, a \in \mathbf{A},\ \forall\, a' \in \mathbf{A}' : a \preceq_{\alpha_{A'}} a' \Rightarrow p(a) \preceq_{\alpha'} p'(a')$.

Using this logical relation, we can re-formulate the safety criteria expressed in Condition *3* of Definition 2.4 as follows.

Property 2.1 *Let* $\alpha_{A'} : [\mathbf{A}; \mathbf{O}] \rightarrow [\mathbf{A}'; \mathbf{O}']$ *be a facet mapping with respect to* $\alpha' = \{\alpha_{B'_i} : \mathbf{B}_i \rightarrow \mathbf{B}'_i\}$. $\forall\, p \in \mathbf{O}\ \exists p' \in \mathbf{O}' :\ p \preceq_{\alpha_{A'}} p'$.

Proof : We need to prove that the safety condition (Condition *3*) in Definition 2.4 is equivalent to the relation $p \preceq_{\alpha_{A'}} p'\ \forall p \in \mathbf{O}$. We only prove the case for closed operators. The proof for open operators 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 $a \preceq_{\alpha_{A'}} a'$, then

   $$\begin{array}{lll} \alpha_{A'}(p(a)) & \sqsubseteq_{A'} & p'(\alpha_{A'}(a)) \quad \text{by the above assumption} \\ & \sqsubseteq_{A'} & p'(a') \qquad\quad \text{monotonicity of } p' \text{ and } a \preceq_{\alpha_{A'}} a' \end{array}$$

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

2. Suppose that $p \preceq_{\alpha_{A'}} p'$, then

$$\begin{aligned} p \preceq_{\alpha_{A'}} p' &\Leftrightarrow \forall a \in \mathbf{A} : p(a) \preceq_{\alpha_{A'}} p(\alpha_{A'}(a)) && \text{since } a \preceq_{\alpha_{A'}} \alpha_{A'}(a) \\ &\Leftrightarrow \forall a \in \mathbf{A} : \alpha_{A'}(p(a)) \sqsubseteq_{A'} p(\alpha_{A'}(a)) && \text{by Definition 2.5} \\ &\Leftrightarrow \forall a \in \mathbf{A} : (\alpha_{A'} \circ p)(a) \sqsubseteq_{A'} (p' \circ \alpha_{A'})(a) \end{aligned}$$

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

This concludes the proof. □

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

2.3 Properties Used in On-Line PPE

This section presents the use of static properties in on-line PPE. We first define the notion of facet by instantiating the abstraction methodology described in Section 2.2. Then, we describe briefly on-line PPE. (The detail specification of on-line PPE semantics is given in Chapter 3.)

2.3.1 Facets

A facet captures symbolic computations performed on the static properties used during on-line PPE. As a result, while a closed operator computes new abstract values, an open operator, when provided with appropriate abstract values, produces constants to be used in partially evaluating expressions. Formally,

Definition 2.6 (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* $\hat{\alpha}_{\widehat{D}} : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ *with respect to* $\hat{\tau}$.

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 2.2 *For any open operator* $p \in \mathbf{O}$ *of arity* n, $\forall \hat{d}_1, \ldots, \hat{d}_n \in \widehat{\mathbf{D}}$ *and* $\forall d_i \in \mathbf{D}$, *if* $d_i \preceq_{\hat{\alpha}_{\widehat{D}}} \hat{d}_i \ \forall\, i \in \{1, \ldots, n\}$, *then*

$$\hat{p}(\hat{d}_1, \ldots, \hat{d}_n) \in \mathbf{Const} \wedge p(d_1, \ldots, d_n) \neq \bot \quad \Rightarrow \quad \hat{p}(\hat{d}_1, \ldots, \hat{d}_n) = \hat{\tau}\ (p(d_1, \ldots, d_n))$$

The proof of this property is trivial, and omitted here. In essence, it states that if an open operator of a facet yields a constant for some abstract values, this constant is the textual representation of the value produced by the concrete operator called with the corresponding concrete values. This is due to the fact that constants are pairwise incomparable. Notice that this equality only holds if the call to the concrete operator terminates. The concrete values d_i are related to the abstract values $\hat{d}_i$ under the logical relation $\preceq_{\hat{\alpha}_{\widehat{D}}}$.

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 following two examples.

The first example of a facet is the vector-size facet taken from the vector algebra as defined in Section 2.1. We have seen part of its definition, and now we present the complete one below. (As mentioned in page 18, function $\mathcal{K}$ turns basic values into constants.)

Example 1 *Vector-size information forms a facet for the vector algebra* $[\mathbf{V}; \mathbf{O}]$.

1. $\widehat{\mathbf{V}} = \{\bot_{\widehat{V}}, s_0, s_1, s_2, \ldots, \top_{\widehat{V}}\}$ *with* $\forall\, \hat{v} \in \widehat{\mathbf{V}} : \bot_{\widehat{V}} \sqsubseteq v \sqsubseteq \top_{\widehat{V}}$.

2. *Abstraction function*

$$\begin{array}{lcl} \hat{\alpha}_{\widehat{V}} & : & \mathbf{V} \rightarrow \widehat{\mathbf{V}} \\ \hat{\alpha}_{\widehat{V}}(v) & = & v = \bot \rightarrow \bot_{\widehat{V}},\ s_{Vec\sharp(v)} \end{array}$$

3. *Closed operators*

$$\widehat{MkVec} : \widehat{\mathbf{Values}} \rightarrow \widehat{\mathbf{V}}$$
$$\widehat{MkVec}\,(c) = (c = \bot_{\widehat{Values}}) \rightarrow \bot_{\widehat{V}},\ (c = \top_{\widehat{Values}}) \rightarrow \top_{\widehat{V}},\ s_{\mathcal{K}(c)}$$
$$\widehat{UpdVec} : \widehat{\mathbf{V}} \times \widehat{\mathbf{Values}} \times \widehat{\mathbf{Values}} \rightarrow \widehat{\mathbf{V}}$$
$$\widehat{UpdVec}\,(\hat{v}, c, r) = (c = \bot_{\widehat{Values}}) \vee (r = \bot_{\widehat{Values}}) \rightarrow \bot_{\widehat{V}},\ \hat{v}$$

4. *Open operators*

$$\widehat{Vec\sharp} : \widehat{\mathbf{V}} \rightarrow \widehat{\mathbf{Values}}$$
$$\widehat{Vec\sharp}\,(\hat{v}) = (\hat{v} = \bot_{\widehat{V}}) \rightarrow \bot_{\widehat{Values}},\ (\hat{v} = s_i) \rightarrow \hat{\tau}(i), \top_{\widehat{Values}}$$
$$\widehat{Vref} : \widehat{\mathbf{V}} \times \widehat{\mathbf{Values}} \rightarrow \mathbf{Values}$$
$$\widehat{Vref}\,(\hat{v}, c) = (\hat{v} = \bot_{\widehat{V}}) \vee (c = \bot_{\widehat{Values}}) \rightarrow \bot_{\widehat{Values}},\ \top_{\widehat{Values}}$$

In the second example, we want to define a sign facet from an integer algebra. A natural 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: when possible, it produces constants which are used to trigger reduction of expressions at partial-evaluation time.

Example 2 *Sign information forms a facet for semantic algebra* $[\mathbf{D}; \mathbf{O}] = [\mathbf{Int};\ \{+, <\}]$.

1. $\widehat{\mathbf{D}} = \{\bot,\ pos,\ zero,\ neg,\ \top\}$ *with* $\forall\, \hat{d} \in \widehat{\mathbf{D}} : \bot \sqsubseteq \hat{d} \sqsubseteq \top$

2. *Abstraction function*

$$\hat{\alpha}_{\widehat{D}} : \mathbf{D} \rightarrow \widehat{\mathbf{D}}$$
$$\hat{\alpha}_{\widehat{D}}\,(d) = \begin{array}{l} d = \bot \rightarrow \bot_{\widehat{D}}, \\ d > 0 \rightarrow pos, \\ d = 0 \rightarrow zero, neg \end{array}$$

3. *Closed operators*

$$\hat{+} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \rightarrow \widehat{\mathbf{D}}$$
$$\hat{+} = \lambda\,(\hat{d}_1,\ \hat{d}_2).\ (\hat{d}_1 = \bot) \vee (\ \hat{d}_2 = \bot) \rightarrow \bot,\ \hat{d}_1 = zero \rightarrow \hat{d}_2,\ \hat{d}_2 = zero \rightarrow \hat{d}_1, \hat{d}_1 \sqcup \hat{d}_2$$

4. Open operators

$$\hat{<} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \rightarrow \widehat{\mathbf{Values}}$$

$$\begin{aligned}\hat{<} = \lambda\,(\hat{d}_1,\ \hat{d}_2).\ & (\hat{d}_1 = \bot) \vee (\hat{d}_2 = \bot) \ \rightarrow\ \bot_{\widehat{Values}},\\ & (\hat{d}_1 = pos) \wedge (\ \hat{d}_2 \in \{neg,\ zero\}) \ \rightarrow\ \hat{\tau}(false),\\ & (\hat{d}_1 = zero) \wedge (\hat{d}_2 = pos) \ \rightarrow\ \hat{\tau}(true),\\ & (\hat{d}_1 = zero) \wedge (\ \hat{d}_2 \in \{neg,\ zero\}) \ \rightarrow\ \hat{\tau}(false),\\ & (\hat{d}_1 = neg) \wedge (\ \hat{d}_2 \in \{pos,\ zero\}) \ \rightarrow\ \hat{\tau}(true),\ \top_{\widehat{Values}}\end{aligned}$$

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

2.3.2 Product of Facets

PPE differs from conventional partial evaluation in two ways: it collects facet information, and it propagates the results of facet computations to all relevant facets. While the latter aspect is described in the PPE model in Chapter 3, the former is captured by the notion of *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 $\hat{\omega}_p$, invokes each facet operation $\hat{p}_i$ with its 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 2.7 (Product of Facets) *Let* $\hat{\alpha}_i : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{D}}_i;\widehat{\mathbf{O}}_i]$ *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* $[\hat{\mathcal{D}};\hat{\Omega}]$, *consists of two components:*

1. *A domain* $\hat{\mathcal{D}} = \widehat{\mathbf{D}}_1 \otimes \cdots \otimes \widehat{\mathbf{D}}_m \cong \prod_{i=1}^{m} \widehat{\mathbf{D}}_i$,

2. *A set of product operators* $\hat{\Omega}$ *such that* $\forall p \in \mathbf{O}\ \exists \hat{\omega}_p \in \hat{\Omega}$:

(a) if $p : \mathbf{D}^n \to \mathbf{D} \in \mathbf{O}$ *is a closed operator, then*

$\hat{\omega}_p : \widehat{\mathcal{D}}^n \to \widehat{\mathcal{D}}$ *and*

$$\hat{\omega}_p = \lambda\ (\hat{\delta}_1, \ldots, \hat{\delta}_n)\ .\ \prod_{i=1}^{m} \hat{p}_i(\hat{\delta}_1^i, \ldots, \hat{\delta}_n^i)$$

(b) otherwise, $p : \mathbf{D}^n \to \mathbf{D}' \in \mathbf{O}$ *is an open operator, then*

$\hat{\omega}_p : \widehat{\mathcal{D}}^n \to \widehat{\mathbf{Values}}$ *and*

$$\hat{\omega}_p = \lambda(\hat{\delta}_1, \ldots, \hat{\delta}_n)\ .\ (\exists j \in \{1, \ldots, m\}\ s.t.\ \hat{d}_j = \perp_{\widehat{Values}}) \to \perp_{\widehat{Values}},$$
$$(\exists j \in \{1, \ldots, m\}\ s.t.\ \hat{d}_j \in \mathbf{Const}) \to \hat{d}_j, \top_{\widehat{Values}}$$
$$where\ \ \hat{d} = \langle \hat{p}_1(\hat{\delta}_1^1, \ldots, \hat{\delta}_n^1), \ldots, \hat{p}_m(\hat{\delta}_1^m, \ldots, \hat{\delta}_n^m) \rangle$$

Domain $\widehat{\mathcal{D}}$ is partially ordered component-wise. Recall that for $d \in \widehat{\mathcal{D}}$, d^i denotes the i^{th} element of d. Smashed product construction is used in constructing product domain; this ensure facet consistency, as explained below.

Although facets of a product are defined independently, the facet values with respect 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 2), and the other indicates its parity (*i.e.*, whether a value is odd or even). Then, a value such as $\langle$*zero*, *odd*$\rangle$ should not be considered a valid facet value since *zero* is an even number. Formally,

Definition 2.8 (Facet Consistency) *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*

$$\bigcap_{i=1}^{m} \{d \in \mathbf{D} \mid d \preceq_{\widehat{\alpha}_i} \hat{\delta}^i\}\ \textit{is neither}\ \emptyset\ \textit{nor}\ \{\perp\}.$$

In the definition, each set of concrete values to be intersected corresponds to a particular facet property; it is defined by the logical relation $\preceq_{\widehat{\alpha}_i}$. Notice that by definition of the relation $\preceq_{\widehat{\alpha}_i}$, the intersection will at least yield the singleton set $\{\perp\}$; therefore such singleton set must not imply consistency. In essence, the above

definition ensures that a product of abstract values represents an actual sub-domain of **D**.

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

We *assume that a program is always specialized with respect to consistent product-of-facet values.*

By the definition of facet it is easy to see that the consistency property is preserved by both open and closed operations. This property contributes to the correctness of the following lemma which states that if there are more than one facet that produce constants, these constants are equal.

Lemma 2.1 *Let* $[\widehat{\mathcal{D}}; \widehat{\Omega}]$ *be a product of facets and* $p \in \mathbf{O}$ *be an open operator,*

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

Proof : Without loss of generality, we consider unary open operators (the argument is noted $\hat{\delta}$). Let $C = \bigcap_{i=1}^{m} \{d \in \mathbf{D} \mid d \preceq_{\widehat{\alpha}_i} \hat{\delta}^i\}$. Since $\hat{\delta}$ is consistent, it is true that $C \neq \emptyset$ and $C \neq \{\perp\}$. Suppose $\exists d \in C$ such that $p(d)$ terminates. Then, by Property 2.2, we have

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

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

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

Lastly, we show below a property about the product operators — its continuity.

Property 2.3 *All operators defined in the product of facets,* $[\widehat{\mathcal{D}}; \widehat{\Omega}]$, *are continuous.*

To prove Property 2.3, we notice that, since every facet domain in a product is of finite height, it suffices to show that the facet operators are monotonic. This can be proven by a case analysis of the different classes of values produced by the operation.

We have seen how properties of interest can be formally introduced via a facet and how facets can 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.

2.3.3 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 conventional partial-evaluation behavior of primitives. It is called the *partial-evaluation facet.* More specifically, whenever it is passed constant arguments, a primitive defined in the partial-evaluation facet will perform the corresponding primitive operation defined in the standard semantics. The partial-evaluation facet is defined as follows:

Definition 2.9 (Partial-Evaluation Facet) *The* partial-evaluation facet *of a semantic algebra* $[\mathbf{D};\mathbf{O}]$ *is defined by the facet mapping* $\hat{\alpha}_{\widehat{Values}} : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{Values}};\widehat{\mathbf{O}}]$ *with respect to* $\hat{\tau}$ *such that:*

1. $\hat{\alpha}_{\widehat{Values}} : \mathbf{D} \rightarrow \widehat{\mathbf{Values}}$
   $\hat{\alpha}_{\widehat{Values}} \equiv \hat{\tau}_D$
2. $\forall\, \hat{p} \in \widehat{\mathbf{O}}$ *of arity* n
   $\hat{p} : \widehat{\mathbf{Values}}^n \rightarrow \widehat{\mathbf{Values}}$
   $\hat{p} = \lambda\ (\hat{d}_1,\ldots,\hat{d}_n)\ .\ \exists i \in \{1,\ldots,n\}\ s.t.\ \hat{d}_i = \perp_{\widehat{Values}} \rightarrow \perp_{\widehat{Values}},$
   $\bigwedge_{i=1}^{n}(\hat{d}_i \in \mathbf{Const}) \rightarrow \hat{\tau}(\mathcal{K}_p[\![p]\!](d_1,\ldots,d_n)), \top_{\widehat{Values}}$
   where $d_i = (\mathcal{K}\ \hat{d}_i)\ \ i \in \{1,\ldots,n\}$

The abstraction function $\hat{\alpha}_{\widehat{Values}}$ corresponds to the abstraction function $\hat{\tau}_D$ defined for domain $\mathbf{D}$; the latter is obtained from the family of abstraction functions $\hat{\tau}$ given in page 18. $\hat{\alpha}_{\widehat{Values}}$ maps a value into its textual representation (that is, a constant). It is easy to verify the following property about the partial-evaluation facet.

Property 2.4 *The partial-evaluation facet (Definition 2.9) is a facet.*

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

We can now discuss on-line PPE.

2.3.4 On-Line PPE

Since the task of an on-line parameterized partial evaluator is to perform partial evaluation, it is natural to assume that the partial-evaluation facet always exists. By convention, it is assigned to the first component of every product of facets. A sum of these product domains is noted $\widehat{\mathcal{SD}}$; each summand corresponds to the domain of a semantic algebra.

Without loss of generality, 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, and available to the parameterized partial evaluator.

To perform PPE, we need to provide, besides the subject program and the program input, a list of product of facets needed in this run. Let PPE_{prog} denotes a parameterized partial evaluator. Its functionality can be informally described as follows:

$$PPE_{prog} \ : \ [\text{Product-of-Facet}] \times \text{Program} \times \text{Input} \ \rightarrow \ \text{Residual-Program}$$

where the symbol $[x]$ denotes a list of items of kind x.

The essential job of a parameterized partial evaluator is to parameterized partially evaluate expressions. This includes constructing residual expressions and computing facet information. Thus, the domain used by the function that partially evaluates expressions is defined as $\mathbf{Exp} \times \widehat{\mathcal{SD}}$, where **Exp** is a flat domain of expressions.

The semantics of PPE is very similar to that of conventional partial evaluation. The details are discussed in Section 3.4. In this section, we investigate how facet

information is used in PPE.

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

- Function $\hat{\mathcal{K}}$ handles constants.
  $\hat{\mathcal{K}}\ [\![c]\!] = \langle [\![c]\!], \langle \hat{\alpha}_{\hat{\mathcal{D}}_1}(d), \ldots, \hat{\alpha}_{\hat{\mathcal{D}}_m}(d) \rangle \rangle \quad where\ d = (\mathcal{K}\ [\![c]\!]) \in \mathbf{D}$

- Function $\hat{\mathcal{K}}_P$ handles primitive operations.
  $\hat{\mathcal{K}}_P\ [\![p^c]\!]\ (\langle e'_1, \hat{\delta}_1 \rangle, \ldots, \langle e'_n, \hat{\delta}_n \rangle) =$
  $(\hat{\delta} = \perp_{\hat{\mathcal{D}}}) \rightarrow \langle \perp_{Exp}, \perp_{\widehat{\mathcal{SD}}} \rangle,\ (\hat{\delta}^1 \in \mathbf{Const}) \rightarrow \langle \hat{\delta}^1, \langle \hat{\alpha}_{\hat{\mathcal{D}}_1}(d), \ldots, \hat{\alpha}_{\hat{\mathcal{D}}_m}(d) \rangle \rangle,$
  $\langle [\![p^c(e'_1, \ldots, e'_n)]\!], \hat{\delta} \rangle$
  $where\ p^c : \mathbf{D}^n \rightarrow \mathbf{D}$
  $\hat{\delta} = \hat{\omega}_{p^c}(\hat{\delta}_1, \ldots, \hat{\delta}_n)$
  $d = \mathcal{K}\ \hat{\delta}^1$
  $\hat{\mathcal{K}}_P\ [\![p^o]\!]\ (\langle e'_1, \hat{\delta}_1 \rangle, \ldots, \langle e'_n, \hat{\delta}_n \rangle) =$
  $(\hat{\delta} = \perp_{\widehat{Values}}) \rightarrow \langle e', \perp_{\widehat{\mathcal{SD}}} \rangle,\ \hat{\delta} \in \mathbf{Const} \rightarrow \langle \hat{\delta}, \langle \hat{\alpha}_{\hat{\mathcal{D}}'_1}(d), \ldots, \hat{\alpha}_{\hat{\mathcal{D}}'_m}(d) \rangle \rangle$
  $\langle e', \langle \top_{\hat{\mathcal{D}}'_1}, \ldots, \top_{\hat{\mathcal{D}}'_m} \rangle \rangle$
  $where\ p^o : \mathbf{D}^n \rightarrow \mathbf{D}'$
  $\hat{\delta} = \hat{\omega}_{p^o}(\hat{\delta}_1, \ldots, \hat{\delta}_n)$
  $d = \mathcal{K}\ \hat{\delta}$
  $e' = [\![p^o(e'_1, \ldots, e'_n)]\!]$

Figure 2.3: $\hat{\mathcal{K}}$ and $\hat{\mathcal{K}}_P$ in the On-Line PPE

Figure 2.3 displays the actions taken when constants and primitive operations are encountered during partial evaluation. For a product of facets $\hat{\mathcal{D}}$, $\hat{\alpha}_{\hat{D}_i}$ denotes the i^{th} abstraction function. We define function $\hat{\mathcal{K}}$ to handle constants, and function $\hat{\mathcal{K}}_P$ to handle primitive operations. Closed and open operators are respectively noted p^c and p^o. Function $\mathcal{K}$ converts a constant to its denotable value, *i.e.*, a basic value in **Values.**

$\hat{\mathcal{K}}$ takes a constant as argument, and invokes the abstraction functions of the corresponding facets to compute the static properties of this constant. It returns

both the constant (as an expression) and the set of static properties.

$\widehat{\mathcal{K}_P}$ accepts a primitive operator and its arguments, and calls the corresponding product operator for computation. For the case of a closed operation, only the first component of the product of facets (the partial-evaluation facet) is able to produce a constant. If a constant is produced, its static properties are re-computed for better accuracy, and the constant is returned as the expression component. Otherwise, a residual expression is returned, and the primitive operation is delayed until run-time.

Treatment of an open operation differs from that of a closed operation in that any facet of the product of facets may produce a constant. Therefore, it is necessary to check all facet components to find such a constant.

Finally, we obtain a conventional on-line partial evaluator if only the partial-evaluation facet is used in each product of facets defined. Thus, PPE is indeed an extension of the conventional one.

Inner-product Example (*Cont.*)

Let us now return to the problem, described in Section 2.1, of partially evaluating an inner-product program with respect to any pair of vectors of size 3. We recall from Section 2.3.4 that an on-line parameterized partial evaluator is informally described as

$$PPE_{prog} \ : \ [\text{Product-of-Facet}] \times \text{Program} \times \text{Input} \ \rightarrow \ \text{Residual-Program}$$

We have defined the vector-size facet in Example 1. There are two facets to be used in replace of the vector algebra during partial evaluation: the partial-evaluation facet and the vector-size facet. These two facets form a product of facets for vector algebra. Referring to the original inner-product program in Figure 2.1, the input to the main function `iprod`, vectors `A` and `B`, will be represented in domain $\mathbf{Exp} \times \widehat{\mathcal{SD}}$ as $\langle \mathtt{A}, \langle \top_{\widehat{Values}}, s_3 \rangle \rangle$ and $\langle \mathtt{B}, \langle \top_{\widehat{Values}}, s_3 \rangle \rangle$ respectively (where `A` and `B` are residual identifiers for `iprod`).

When partially evaluating `iprod`, the vector-size property is used to obtain the size of vector `A`. Specifically, we have $\widehat{Vec\sharp}\ (s_3) = \hat{\tau}(3)$. The induction variable of `dotProd`, `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 2.4, which is as desired. Finally, 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 made residual.

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

Figure 2.4: Residual Program for Inner-Product Computation
Residual program produced by partially evaluating the inner-product program with respect to any pair of vectors of size 3.

2.4 Properties Used in Off-Line PPE

As discussed in Chapter 1, 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 on, for example, constant 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 partially evaluating recursive calls to a function such that the staticity of the arguments remain the same at each call.

In conventional partial evaluation efficiency is achieved by an *off-line* strategy which splits the partial-evaluation phase into binding-time analysis and specialization [Jones *et al.*, 1989,Bondorf, 1990,Consel, 1990b]. In particular, binding-time analysis only computes static/dynamic property. In off-line PPE, we generalize binding-time

analysis to *facet analysis*: an analysis that statically computes user-defined static properties. Consequently, the task of program specialization reduces to following the information yielded by facet analysis.

In this section, we investigate the use of static properties in off-line PPE. Since the properties used by an off-line specializer are similar to that used by the on-line counterpart, we concentrate on the use of static properties by facet analysis. In this respect, we follow the approach used in introducing properties into on-line PPE: we introduce the concept of abstract facet in Section 2.4.1, describe the product of abstract facets in Section 2.4.2, define the binding-time facet in Section 2.4.3, and lastly, describe briefly facet analysis in Section 2.4.4.

2.4.1 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 2.10 (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* $\tilde{\tau}$.

Recall from page 19 that $\tilde{\tau}$ is an abstraction function mapping from $\widehat{\mathbf{D}}$ to $\widetilde{\mathbf{D}}$. The

use of facet mapping in the definition ensures the following property about the open operators of an abstract facet.

Property 2.5 *For any open operator* $\hat{p} \in \widehat{\mathbf{O}}$ *of arity* n, $\forall \tilde{d}_1, \ldots, \tilde{d}_n \in \widetilde{\mathbf{D}}$ *and* $\forall \hat{d}_i \in \widehat{\mathbf{D}}$, *if* $\hat{d}_i \preceq_{\widetilde{\alpha}_{\widetilde{D}}} \tilde{d}_i$ *for* $i \in \{1, \ldots, n\}$, *then*

$$(\tilde{p}(\tilde{d}_1, \ldots, \tilde{d}_n) = \mathit{static}) \;\Rightarrow\; \hat{p}(\hat{d}_1, \ldots, \hat{d}_n) \sqsubseteq_{\widehat{Values}} c \text{ with } c \in \mathbf{Const}.$$

Proof : By the safety condition for facet mapping (*i.e.*, condition *3* of facet mapping), we must have

$$\tilde{\tau}(\hat{p}(\hat{d}_1, \ldots, \hat{d}_n)) \;\sqsubseteq_{\widetilde{Values}} \tilde{p}(\tilde{d}_1, \ldots, \tilde{d}_n) = \mathit{static}.$$

From definition of $\tilde{\tau}$ (page 19), we have $\forall x \in \widehat{\mathbf{Values}}$, $\tilde{\tau}(x) \sqsubseteq_{\widetilde{Values}} \mathit{static} \;\Rightarrow\; x \in \mathbf{Const} \cup \{\perp_{\widehat{Values}}\}$. Therefore $\hat{p}(\hat{d}_1, \ldots, \hat{d}_n) \in \mathbf{Const} \cup \{\perp_{\widehat{Values}}\}$. □

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.

In the case of partially evaluating the inner-product program (in Section 2.1) using off-line PPE, a possible static property that we may like to compute in the analysis phase will be the availability of vector-size information during specialization. This can be described as a vector-size abstract facet derived from the vector-size facet (Example 1). This enables us to determine, prior to specialization, whether vector-size computation can produce constants. The vector-size abstract facet is defined below:

Example 3 *Information about Vector-size availability forms an abstract facet with respect to the vector-size facet* $[\widehat{\mathbf{V}}; \widehat{\mathbf{O}}]$.

1. $\widetilde{\mathbf{V}} = \{s, d\}$ *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*

$\tilde{\alpha}_{\widetilde{V}} : \widehat{\mathbf{V}} \rightarrow \widetilde{\mathbf{V}}$
$\tilde{\alpha}_{\widetilde{V}}(\hat{v}) = \hat{v} = \perp_{\widehat{V}} \rightarrow \perp_{\widetilde{V}}, \quad \hat{v} = \top_{\widehat{V}} \rightarrow d, \; s$

3. *Closed operators*

$\widetilde{MkVec} : \widetilde{\mathbf{Values}} \rightarrow \widetilde{\mathbf{V}}$
$\widetilde{MkVec}\,(i) = (i = \perp_{\widetilde{Values}}) \rightarrow \perp_{\widetilde{V}}, \; (i = dynamic) \rightarrow d, \; s$
$\widetilde{UpdVec} : \widetilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \times \widetilde{\mathbf{Values}} \rightarrow \widehat{\mathbf{V}}$
$\widetilde{UpdVec}\,(\tilde{v}, i, k) = (i = \perp_{\widetilde{Values}}) \vee (r = \perp_{\widetilde{Values}}) \rightarrow \perp_{\widetilde{V}}, \; \tilde{v}$

4. *Open operators*

$\widetilde{Vec\sharp} : \widetilde{\mathbf{V}} \rightarrow \widetilde{\mathbf{Values}}$
$\widetilde{Vec\sharp}\,(\tilde{v}) = (\tilde{v} = \perp_{\widetilde{V}}) \rightarrow \perp_{\widetilde{Values}}, \; (\tilde{v} = s) \rightarrow \; static, \; dynamic$
$\widetilde{Vref} : \widetilde{\mathbf{V}} \times \widetilde{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}}$
$\widetilde{Vref}(\tilde{v}, i) = (\tilde{v} = \perp_{\widetilde{V}}) \vee (i = \perp_{\widetilde{Values}}) \rightarrow \perp_{\widetilde{Values}}, \; dynamic$

We don't need to restrict the static properties used in the analysis phase to be one with domain of three (or even two) elements. In fact, the definition of abstract facet is general enough that its domain can virtually be of arbitrary size. However, to ensure the termination of the analysis, all domains used in the analysis need to be of bounded size. In example 2, we define a sign facet. We can also define a sign abstract facet which, at analysis time, determines whether sign computation can produce constants. The size abstract facet's domain has more than three elements.

Example 4 *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 2)

2. $\tilde{\alpha}_{\widetilde{D}}$ *is 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.*

$$\begin{aligned}
\tilde{<} \;&:\; \widetilde{\mathbf{D}} \times \widetilde{\mathbf{D}} \rightarrow \widetilde{\mathbf{Values}} \\
\tilde{<} \;&=\; \lambda\,(a,b).\; a = \bot \vee b = \bot \rightarrow \bot_{\widetilde{Values}}, \\
&\qquad a = pos \wedge (\, b \in \{neg,\ zero\}) \rightarrow static, \\
&\qquad a = zero \wedge b = pos \rightarrow static, \\
&\qquad a = zero \wedge (\, b \in \{neg,\ zero\}) \rightarrow static, \\
&\qquad a = neg \wedge (\, b \in \{pos,\ zero\}) \rightarrow static,\ dynamic
\end{aligned}$$

The above two examples reinforce the idea that both facets and abstract facets are defined under one abstraction methodology. As such, the user will not experience any "cultural shock" when shifting from one partial-evaluation strategy to another. The similarities in modeling static properties do not end here; even the grouping and the use of static properties are similar, as will be illustrated in the next few sections.

2.4.2 Product of Abstract Facets

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

Definition 2.11 (Product of Abstract Facets) *Let* $\tilde{\alpha}_i : [\widehat{\mathbf{D}}_i; \widehat{\mathbf{O}}_i] \rightarrow [\widetilde{\mathbf{D}}_i; \widetilde{\mathbf{O}}_i]$ *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* $[\tilde{\mathcal{D}}, \tilde{\Omega}]$, *consists of two components:*

1. *A domain* $\tilde{\mathcal{D}} = \prod_{i=1}^{m} \widetilde{\mathbf{D}}_i$;

2. *A set of product operators* $\tilde{\Omega}$ *such that* $\forall p \in \mathbf{O}\ \exists \tilde{\omega}_p \in \tilde{\Omega}$:

   (a) *if* $p : \mathbf{D}^n \rightarrow \mathbf{D}$ *is a closed operator, then*

   $\tilde{\omega}_p : \tilde{\mathcal{D}}^n \rightarrow \tilde{\mathcal{D}}$ *and*

   $\tilde{\omega}_p = \lambda\ (\tilde{\delta}_1, \ldots, \tilde{\delta}_n).\ \prod_{i=1}^{m} \tilde{p}_i(\tilde{\delta}_1^i, \ldots, \tilde{\delta}_n^i)$

   (b) *otherwise,* $p : \mathbf{D}^n \rightarrow \mathbf{D}'$ *is an open operator, then*

   $\tilde{\omega}_p : \tilde{\mathcal{D}}^n \rightarrow \widetilde{\mathbf{Values}}$ *and*

   $$\begin{aligned}
   \tilde{\omega}_p = \;& \lambda(\tilde{\delta}_1, \ldots, \tilde{\delta}_n)\ .\ (\exists j \in \{1, \ldots, m\}\ s.t.\ \tilde{d}_j = \bot_{\widetilde{Values}}) \rightarrow \bot_{\widetilde{Values}}, \\
   & \qquad (\exists j \in \{1, \ldots, m\}\ s.t.\ \tilde{d}_j = static) \rightarrow static, dynamic \\
   & \quad where\ \tilde{d} = \langle \tilde{p}_1(\tilde{\delta}_1^1, \ldots, \tilde{\delta}_n^1), \ldots, \tilde{p}_m(\tilde{\delta}_1^m, \ldots, \tilde{\delta}_n^m) \rangle
   \end{aligned}$$

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. The following lemma expresses the fact about the product operators, just like Lemma 2.3. Its proof is similar too.

Property 2.6 *All operators defined in the product of abstract facets,* $[\widetilde{\mathcal{D}}; \widetilde{\Omega}]$, *are continuous.*

2.4.3 Binding-time Facet

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

Definition 2.12 (Binding-Time Facet) *The* binding-time facet *of a partial-evaluation facet* $[\widehat{\mathbf{Values}}; \widehat{\mathbf{O}}]$ *is defined by the facet mapping* $\tilde{\alpha}_{\widetilde{Values}} : [\widehat{\mathbf{Values}}; \widehat{\mathbf{O}}] \rightarrow [\widetilde{\mathbf{Values}}; \widetilde{\mathbf{O}}]$ *with respect to* $\tilde{\tau}$ *such that:*

1. $\tilde{\alpha}_{\widetilde{Values}} : \widehat{\mathbf{Values}} \rightarrow \widetilde{\mathbf{Values}}$ *and*
   $\tilde{\alpha}_{\widetilde{Values}} \equiv \tilde{\tau}$ (defined in page 19)
2. $\forall\, \tilde{o} \in \widetilde{\mathbf{O}}$ *of arity* n
   $\tilde{o} : \widetilde{\mathbf{Values}}^n \rightarrow \widetilde{\mathbf{Values}}$ *and*
   $\tilde{o} = \lambda\, (\tilde{d}_1, \ldots, \tilde{d}_n)\ .\ \exists j \in \{1, \ldots, n\}\ s.t.\ \tilde{d}_j = \perp_{\widetilde{Values}} \rightarrow \perp_{\widetilde{Values}},$
   $\bigwedge_{i=1}^{n} (\tilde{d}_i = static) \rightarrow static,\ dynamic$

It is easy to verify the following property about the binding-time facet.

Property 2.7 *The binding-time facet (Definition 2.12) is an abstract facet.*

Not surprisingly, Definition 2.12 captures the primitive operations of conventional binding-time analysis. As a result, not only does a facet analysis compute user-defined abstract values, but it also computes binding-time values in the way that a binding-time analysis does.

2.4.4 Facet Analysis

We are now ready to discuss facet analysis. It is essentially a conventional binding-time analysis, as described in [Sestoft, 1985] for example, extended to compute abstract-facet information. Analogous to the definition of on-line PPE, 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. Detailed discussion appears in Chapter 3. In this section, we investigate how abstract-facet information is used in facet analysis.

Figure 2.5 displays the actions taken when constant and primitive operations are encountered during facet analysis. We define function $\widetilde{\mathcal{K}}$ to handle constants, and function $\widetilde{\mathcal{K}}_P$ to handle primitive operations. Closed and open operators are respectively noted p^c and p^o.

$\widetilde{\mathcal{K}}$ takes a constant as argument, invokes the abstraction functions of the corresponding facets *and* abstract facets to compute the static properties of this constant. (Facets are needed as well as abstract facets because the latter is defined as an abstraction of the former.) It returns the set of static properties used at facet-analysis time.

Analogous to function $\widehat{\mathcal{K}}_P$ in Figure 2.3, $\widetilde{\mathcal{K}}_P$ accepts a primitive operator and its arguments, and calls the corresponding abstract product operator for computation.

Inner-product Example (*Cont.*)

Let us return to our example of partially evaluating the inner-product program with respect to any pair of vectors of size 3. In the analysis phase, we may want to deal with the availability of vector-size information, and we therefore use the vector-size abstract facet defined in Example 3. In our case, the actual value of both input vectors are dynamic but their sizes are known. Recall that besides the abstract **Size** facet, the binding-time facet (Definition 2.12) is also defined. Both parameters of

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

- Function $\tilde{\mathcal{K}}$ handles constants.
  $\tilde{\mathcal{K}}\ [\![c]\!] = \langle \tilde{\Gamma}_1(d), \ldots, \tilde{\Gamma}_m(d) \rangle$ *where* $\tilde{\Gamma}_i = \tilde{\alpha}_{\widetilde{\mathcal{D}}_i} \circ \hat{\alpha}_{\widehat{\mathcal{D}}_i}$ *and* $d = (\mathcal{K}\ c)$

- Function $\tilde{\mathcal{K}}_P$ handles primitive operations.
  $\tilde{\mathcal{K}}_P\ [\![p^c]\!]\ (\tilde{\delta}_1, \ldots, \tilde{\delta}_n) = \tilde{\omega}_{p^c}(\tilde{\delta}_1, \ldots, \tilde{\delta}_n)$ *where* $p^c : \mathbf{D}^n \rightarrow \mathbf{D}$
  $\tilde{\mathcal{K}}_P\ [\![p^o]\!]\ (\tilde{\delta}_1, \ldots, \tilde{\delta}_n) = \tilde{\delta} = \perp_{\widetilde{Values}} \rightarrow \perp_{\widetilde{SD}}, \langle \tilde{\delta}, \top_{\widetilde{D}'_2}, \ldots, \top_{\widetilde{D}'_m} \rangle$
  where $p^o : \mathbf{D}^n \rightarrow \mathbf{D}'$
  $\tilde{d} = \tilde{\omega}_{p^o}(\tilde{\delta}_1, \ldots, \tilde{\delta}_n)$

Figure 2.5: $\tilde{\mathcal{K}}$ and $\tilde{\mathcal{K}}_P$ in Facet Analysis

`iprod` will then be bound to the pair of abstract values ⟨`dynamic, s`⟩. Consequently, the binding-time value of variable `n` is `static`. Thus, 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 PPE; however, these reductions have been determined statically.

Figure 2.6 displays the information yielded by facet analyzing the inner-product program when only the vector sizes are static. 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 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 the abstract syntax tree rooted by the open operation `Vec♯`.) Consequently, only partial-evaluation-facet computation is required for `dotProd` at specialization time.

This contrasts with on-line PPE of the inner-product program where the size-facet

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

Figure 2.6: Abstract-Facet Information after Facet Analysis

computation is performed each time function `dotProd` is invoked.

2.5 Discussion

We have defined an abstraction methodology called facet mapping that models the static properties for both the on-line and off-line levels of partial evaluation. Facet mapping is general enough to model conventional partial-evaluation behavior and binding-time behavior of primitive operations. As such, we can naturally extend conventional partial evaluation to PPE, at both the on-line and off-line levels.

So far, we have laid out the techniques for defining as well as utilizing static properties. Throughout the discussion, we have linked the framework closely to conventional partial evaluation. Certainly, it is possible to envision various extension to this framework so that more accurate facet information can be derived. Several possible extensions are listed below.

2.5.1 Encoded Concrete Values

A feature of PPE is the ability to derive constants from the facet computation in addition to the usual way of producing constants via static-expression evaluation. Recall that constants are textual representation of concrete values. In defining a facet, it is possible that some concrete values are encoded as elements of the facet domain. For instance, the symbolic value *zero* in the sign facet (Example 2) corresponds to the constant 0 in $\widehat{\textbf{Values}}$ and the concrete value 0 in the integer domain. A partial evaluator cannot establish the relationship between the encoded concrete values and their corresponding concrete values. Currently, such a relationship is only realized (and used) through the evaluation of open operators. This is because open operations may evaluate to some constants in $\widehat{\textbf{Values}}$ — the domain of the partial-evaluation facet.

Similarly, in the analysis phase of off-line PPE, static binding-time information may be derived from the abstract-facet values used. For instance, we may wish that facet analysis recognizes the fact that the abstract value *zero* in the sign *abstract* facet is static information that can be utilized in the specialization phase.

In this section, we present a means for a partial evaluator to recognize these encoded concrete values and use them as constants when they appear, without resorting to open operations. Hence, *zero* in the sign facet above will be recognized as constant 0 by the partial evaluator, and as *static* by the facet analysis.

Why Concretization Function Is Not Adequate?

In abstract interpretation, a concrete domain and an abstract domain are related by a pair of functions α and γ. Function α is called the *abstraction function*; it assigns an abstract value to each concrete value in the concrete domain. Function γ is called the *concretization function*; it maps each value in the abstract domain to a set of concrete values. As such, it appears that concrete values (and therefore constant values) can be obtained by first applying γ, then extracting the concrete values from

the resulting set. In the case of defining a facet, although it may be possible to derive function α from the abstraction function defined for the facet, using a concretization function to obtain concrete values from an abstract value is more problematic. Since the co-domain of γ is in general the powerset (or powerdomain, to be exact) of the underlying concrete domain, computation using a concretization function may not terminate in general. In the case of sign facet, we may define $\gamma(zero) = \{0\}$, and 0 can be retrieved from the result of application. However, when we apply γ to the other non-bottom facet information (like *pos*, *neg*), the computation will not terminate since their corresponding values in the concrete domain form an infinite set.

A variant of the concretization function may appear to answer our question: Let $\overline{\alpha} : [\mathbf{D}; \mathbf{O}] \rightarrow [\overline{\mathbf{D}}; \overline{\mathbf{O}}]$ be a facet mapping with respect to $\overline{\tau}$, we define a "concretization function" $\overline{\gamma} : \overline{\mathbf{D}} \rightarrow \overline{\mathbf{Values}}$ such that

$$\forall\, d \in \mathbf{D},\ \ \overline{\gamma}\,(\overline{\alpha}\,(d))\ \sqsubseteq_{\overline{Values}}\ \overline{\tau}\,(d).$$

$\overline{\gamma}$ as defined may function as a decoder for the encoded concrete values during on-line PPE, but it does not work in the analysis phase of off-line PPE. For instance, given the sign abstract facet, if we chose $\tilde{\gamma} : \widetilde{\mathbf{D}} \rightarrow \widetilde{\mathbf{Values}}$ to map $\top_{\widetilde{D}}$ to value *static*, then $\tilde{\gamma}$ can be defined to satisfy the criteria

$$\forall\, \hat{d} \in \widehat{\mathbf{D}},\ \ \tilde{\gamma}\,(\tilde{\alpha}\,(\hat{d}))\ \sqsubseteq_{\widetilde{Values}}\ \tilde{\tau}\,(\hat{d}).$$

Unfortunately, value *static* derived from $\tilde{\gamma}$ does not reflect the fact that the underlying value in domain $\widehat{\mathbf{D}}$ is a constant.

Concrete-value Decoder

What we need is a terminating function that maps those, and only those encoded concrete values (in the facet domain) to their corresponding constant values. We call this function the *concrete-value decoder*. It is defined as follows:

Definition 2.13 (Concrete-value Decoder) *Let* $\hat{\alpha}_{\widehat{D}} : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ *be a facet mapping defined for semantic algebra* $[\mathbf{D};\mathbf{O}]$ *with respect to* $\hat{\tau}$. *A* concrete-value decoder *for the facet* $[\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$ *is a continuous function* $\hat{\mu} : \widehat{\mathbf{D}} \rightarrow \widehat{\mathbf{Values}}$ *such that there exists a set* $\hat{S} \subseteq \widehat{\mathbf{D}}$ *and*

1. $\forall\, \hat{d} \in \widehat{\mathbf{D}},\ \ \hat{\mu}\,(\hat{d}) \in \mathbf{Const} \ \Leftrightarrow\ \ \hat{d} \in \hat{S}$;

2. $\forall\, d \in \mathbf{D},\ \forall \hat{s} \in \hat{S},\ \ \hat{\alpha}_{\widehat{D}}(d) = \hat{s} \ \Rightarrow\ \ \hat{\mu}\,(\hat{\alpha}_{\widehat{D}}\,(d)) = \hat{\tau}\,(d)$.

We call the set $\hat{S}$ the *concrete-value set.*

The first requirement for function $\hat{\mu}$ implies that elements *not* in $\hat{S}$ are mapped to either $\bot_{\widehat{Values}}$ or $\top_{\widehat{Values}}$ in $\widehat{\mathbf{Values}}$. The second requirement ensures the elements in $\hat{S}$ are indeed encoded concrete values. The definition of $\hat{\mu}$ gives the user the flexibility of choosing a set of constants to be recognized by the partial evaluator.

We can extend the definition of a facet to include the concrete-value decoder. This is manifested using the example for sign facet.

Example 5 *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*

   $\hat{\alpha}_{\widehat{D}} \quad : \quad \mathbf{D} \rightarrow \widehat{\mathbf{D}}$
   $\hat{\alpha}_{\widehat{D}}\,(d) \ = \ d = \bot \rightarrow \bot_{\widehat{D}},\ d > 0 \rightarrow pos,\ d = 0 \rightarrow zero,\ neg$
   Notice that zero is an encoded concrete value for 0.

3. $\hat{\mu}$ *is defined as follows:*

   $\hat{\mu} \quad : \quad \widehat{\mathbf{D}} \rightarrow \widehat{\mathbf{Values}}$
   $\hat{\mu}(\hat{d}) \ = \ \hat{d} = \bot \rightarrow \bot, \hat{d} = zero \rightarrow \hat{\tau}(0),\ \top$

4. $\widehat{\mathbf{O}}$, *the set of primitives, is defined in Example 2.*

Lastly, We redefine a facet mapping to be $\langle \hat{\alpha}, \hat{\mu} \rangle : [\mathbf{D};\mathbf{O}] \rightarrow [\widehat{\mathbf{D}};\widehat{\mathbf{O}}]$.

Using Concrete-value Decoder

In defining a facet mapping to be a pair $\langle \hat{\alpha}, \hat{\mu} \rangle$, any property information useful to the partial evaluator can be flown out of the facet, either via open operations or by applying the concrete-value decoder. For a partial evaluator to accept this new channel of information, we modify the definition of product of facets as follows:

Definition 2.14 (Enhanced Product of Facets) *Let* $\langle \hat{\alpha}_i, \hat{\mu}_i \rangle : [\mathbf{D}; \mathbf{O}] \rightarrow [\widehat{\mathbf{D}}_i; \widehat{\mathbf{O}}_i]$ *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* $\widehat{\mathcal{D}} = \widehat{\mathbf{D}}_1 \otimes \ldots \otimes \widehat{\mathbf{D}}_m \cong \prod_{i=1}^{m} \widehat{\mathbf{D}}_i$.

2. *A set of product operators* $\widehat{\Omega}$ *such that* $\forall p \in \mathbf{O}\ \exists \hat{\omega}_p \in \widehat{\Omega}$:

   (a) *if* $p : \mathbf{D}^n \rightarrow \mathbf{D} \in \mathbf{O}$ *is a closed operator, then*

   $\hat{\omega}_p : \widehat{\mathcal{D}}^n \rightarrow \widehat{\mathcal{D}}$ *and*

   $$\hat{\omega}_p = \lambda\ (\hat{\delta}_1, \ldots, \hat{\delta}_n)\ .\ (\exists j \in \{1, \ldots, m\}\ s.t.\ \hat{\mu}_j(\hat{\delta}^j) \in \mathbf{Const}) \rightarrow \langle \hat{\alpha}_{\widehat{\mathcal{D}}_1}(d),\ \ldots, \hat{\alpha}_{\widehat{\mathcal{D}}_m}(d) \rangle,\ \prod_{i=1}^{m} \hat{\delta}^i$$
   $$where\ \prod_{i=1}^{m} \hat{\delta}^i = \prod_{i=1}^{m} \hat{p}_i(\hat{\delta}_1^i, \ldots, \hat{\delta}_n^i)$$
   $$d = \mathcal{K}(\hat{\mu}_j(\hat{\delta}^j))$$

   (b) *otherwise,* $p : \mathbf{D}^n \rightarrow \mathbf{D}' \in \mathbf{O}$ *is an open operator, then*

   $\hat{\omega}_p : \widehat{\mathcal{D}}^n \rightarrow \widehat{\mathbf{Values}}$ *and*

   $$\hat{\omega}_p = \lambda(\hat{\delta}_1, \ldots, \hat{\delta}_n)\ .\ (\exists j \in \{1, \ldots, m\}\ s.t.\ \hat{\delta}_j = \perp_{\widehat{Values}}) \rightarrow \perp_{\widehat{Values}},$$
   $$(\exists j \in \{1, \ldots, m\}\ s.t.\ \hat{\delta}_j \in \mathbf{Const}) \rightarrow \hat{\delta}_j, \top_{\widehat{Values}}$$
   $$where\ \hat{\delta} = \langle \hat{p}_1(\hat{\delta}_1^1, \ldots, \hat{\delta}_n^1), \ldots, \hat{p}_m(\hat{\delta}_1^m, \ldots, \hat{\delta}_n^m) \rangle$$

PPE remains intact since the use of encoded concrete values only occurs at the level of product of facets.

Lastly, the definition of abstract facet needs to be enhanced to capture the concrete-value-decoder operation. This means that during specialization, when an encoded

concrete value is mapped to a constant by the decoder, it may be captured in the analysis phase by mapping the abstract version of this encoded concrete value to value *static* in the abstract facet. The *abstract concrete-value decoder* of an abstract facet makes reference to the underlying facet, and is defined as follows:

Definition 2.15 (Abstract Concrete-value Decoder) *Given a facet defined by the* $\langle \hat{\alpha}, \hat{\mu} \rangle : [\mathbf{D}; \mathbf{O}] \rightarrow [\widehat{\mathbf{D}}; \widehat{\mathbf{O}}]$. *Let* $\tilde{\alpha}_{\widetilde{D}} : [\widehat{\mathbf{D}}; \widehat{\mathbf{O}}] \rightarrow [\widetilde{\mathbf{D}}; \widetilde{\mathbf{O}}]$ *be a facet mapping defined with respect to* $\tilde{\tau}$. *An* abstract concrete-value decoder *for the abstract facet* $[\widehat{\mathbf{D}}; \widehat{\mathbf{O}}]$ *is a continuous function* $\tilde{\mu} : \widetilde{\mathbf{D}} \rightarrow \widetilde{\mathbf{Values}}$ *such that there exists a set* $\tilde{S} \subseteq \widetilde{\mathbf{D}}$ *and*

1. $\forall\, \tilde{d} \in \widetilde{D},\ \ \tilde{\mu}\,(\tilde{d}) \;=\; static \;\;\Leftrightarrow\;\; \tilde{d} \in \tilde{S};$

2. $\forall\, \hat{d} \in \widehat{\mathbf{D}},\ \forall \tilde{s} \in \tilde{S},\ \ \tilde{\alpha}_{\widetilde{D}}(\hat{d}) \;=\; \tilde{s} \;\;\Rightarrow\;\; \tilde{\mu}\,(\tilde{\alpha}_{\widetilde{D}}\,(\hat{d})) \;=\; \tilde{\tau}\,(\hat{\mu}\,(\hat{d})).$

Again, we can enhance the definition of product of abstract facets so that the availability of those constants that can be detected in the analysis phase are computed either via open operations or via abstract concrete-value decoder.

2.5.2 Refining Domain $\widetilde{\mathbf{Values}}$

Traditionally, 3-point (or 2-point) binding-time domain is used by binding-time analysis. Since some constants (concrete values) may appear in any program, it is conceivable that certain amount of constant propagation/folding (and also static computation) be performed in the analysis phase. This requires a new domain which also includes constant values, and a new set of operations that operates on constants. However, experience in partial evaluation has indicated that the additional static computation gained from constant propagation in the analysis phase may not be abundant enough to warrant the construction of more complicated domain.

In an attempt to construct our framework based on the conventional partial evaluation, we define $\widetilde{\mathbf{Values}}$ to be the 3-point domain. However, in the presence of a large variety of static properties, static information may frequently be produced

from abstract-facet computations. Therefore, the use of a binding-time domain *with* constants may become ever more justifiable. In fact, in an implementation of PPE done in CMU, Colby and Lee enhanced the domain $\widetilde{\mathbf{Values}}$ to include constant values [Colby and Lee, 1991]. More experiments are still needed to observe the effect of using this enhanced domain.

Care must be taken when enhancing $\widetilde{\mathbf{Values}}$ to include constants. Since it is necessary to ensure the termination of facet analysis, all domains used in the analysis phase are to be of finite size. Thus, only a finite number of constants can be included in $\widetilde{\mathbf{Values}}$. Let $\mathbf{Const}' \subset \mathbf{Const}$ be the finite set of constants included in $\widetilde{\mathbf{Values}}$, the ordering of the elements in $\widetilde{\mathbf{Values}}$ will be

$$\forall\, c \in \mathbf{Const}',\ \ \bot_{\widetilde{Values}} \sqsubseteq c \sqsubseteq \mathit{static} \sqsubseteq \mathit{dynamic}.$$

Since domain $\widetilde{\mathbf{Values}}$ is used as co-domain of the open operators of any abstract facet, we would expect the definition of these open operators be modified accordingly.

Chapter 3

Semantic Specifications and Correctness Proofs

We have seen in Chapter 2 how we can model a static property by instantiating a facet mapping. In this chapter, we shift our attention to PPE itself, and provide the semantic specifications and correctness proofs of both on-line and off-line PPE. Since PPE is a natural extension of conventional partial evaluation, correctness properties about the former can always be restated to fit the latter. In this chapter, we first describe the existing theoretical work on conventional partial evaluation; we then present our approach and techniques used in building the theoretical foundation of PPE.

3.1 Overview

In Section 1.1, we stated the correctness criterion of a partial evaluation as follows:

> Suppose that $P(x, y)$ is a program with two arguments, whose first argument x is known, but whose second argument y is unknown. Specialization of $P(x, y)$ yields a residual program $P_x(y)$ such that:
>
> $$\forall y,\ P(x, y) = P_x(y) \tag{3.1}$$
>
> provided the evaluation of both $P(x, y)$ and $P_x(y)$ terminates.

Regardless of the strategy used, partial evaluation is a non-trivial process, it involves numerous program transformations. Therefore, proving the correctness of this process must go beyond the extensional criterion given by Equation 3.1; it must be based on the semantics of partial evaluation. This approach should also provide the user with a better understanding of the process.

Several works on proving the correctness of conventional partial evaluation have appeared in the literature recently, all dedicated to off-line partial evaluation. In particular, Gomard in [Gomard, 1992] defines a denotational semantics of a specializer for lambda calculus,[1] together with its correctness proof. However, the specializer is limited to monovariant specialization. (That is, every function in a program can have at most one specialized version created during specialization). In [Launchbury, 1990], Launchbury defines in a denotational style a binding-time analysis and proves its correctness with respect to the standard semantics. He also shows that his result corresponds to the notion of *uniform congruence*, a restrictive version of the congruence criterion for binding-time analysis defined by Jones [Jones, 1988]. However, since the correctness proofs are done with respect to the standard semantics, they do not provide any insight as to how binding-time properties are related to the partial-evaluation process, and more specifically to that of on-line partial evaluation.

The work described in this chapter is distinct from the existing ones in two aspects: First, it provides a correctness proof for *polyvariant* specialization (that is, a function in a program can have more than one specialized version created during specialization); second, it adopts a *uniform approach* for both defining and proving the correctness of on-line and off-line PPE semantics.

3.1.1 Structure of the Semantics

In polyvariant specialization, when a function call is suspended (*i.e.*, the call is not to be unfolded during partial evaluation), a specialized version of the function is created;

[1]The binding-time information are provided by the user, and therefore its derivation is not included in the semantics.

it is this suspended function call that characterizes the specialized function. During partial evaluation, if all suspended function calls are collected, they characterize the residual program, and can actually be used to construct the latter.

This observation prompts us to specify PPE semantics in terms of collecting interpretation, as described in [Hudak and Young, 1991]. (The resulting semantics is also similar to the minimal-function-graph (MFG) semantics [Jones and Mycroft, 1986].) Consequently, as is the case with collecting interpretation, the PPE semantics consists of two functions: the *local semantic function* (or the standard semantic function, using the terminology of [Hudak and Young, 1991]) describes partial evaluation of expressions. The *global semantic function* (correspondingly, the collecting interpretation) describes the collection of *specialization patterns.* (A specialization pattern contains information, obtained from a function call, which characterizes the corresponding specialized function generated.)

3.1.2 Uniform Approach for Defining and Proving the Correctness of PPE Semantics

A uniform approach to defining and proving the correctness of both on-line and off-line PPE semantics enables us to define the relationship between these two levels of partial evaluation. Furthermore, it provides a basis for applying techniques from one level to the other. The uniformity of our approach is based on the following two techniques:

1. **Factorized Semantics:** We define a *core semantics* [Jones and Muchnick, 1976,Jones and Nielson, 1990] which consists of semantic rules, and uses some uninterpreted domain names and combinator names (Section 3.2). This semantics forms the basis for all the semantic specifications defined in this chapter. Specifically, by providing a specific interpretation to the domains and combinators of the core semantics, we obtain an instrumented semantics which extends the standard semantics to capture all function applications performed during

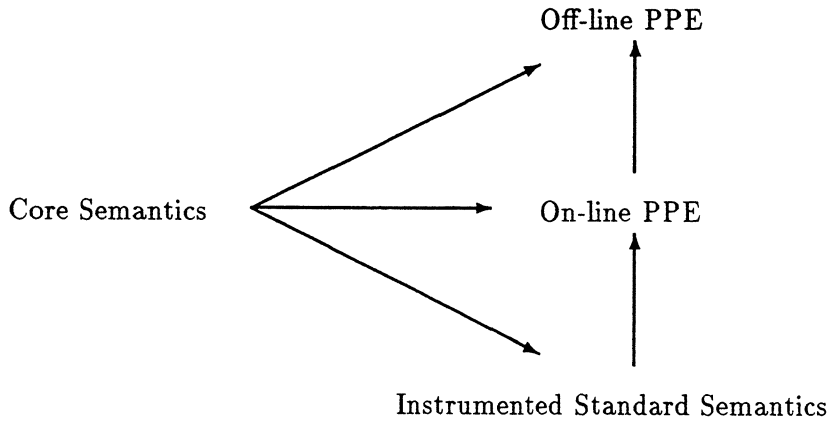

Figure 3.1: Factorized Semantics and Logical Relations

Factorized semantics enable the instantiation of various semantics of interest from the core semantics. Logical relations relate two adjacent levels of semantics.

program execution (Section 3.3). Using other interpretations, we define an on-line PPE semantics (Section 3.4), a facet analysis and a specialization semantics (Section 3.5) respectively. The advantage of using a factorized semantics is that different instances can be related at the level of domain definitions and combinator definitions.

2. **Logical Relations:** We use the technique of *logical relations* [Jones and Nielson, 1990,Abramsky, 1990,Mizuno and Schmidt, 1990] to prove the correctness of PPE semantics. Logical relations are defined (1) to relate on-line PPE semantics to instrumented semantics, and (2) to relate facet analysis to on-line semantics. Since all these semantics are instantiated from the same core semantics, their relations can be defined locally by relating their domains and combinators. The resulting proofs thus conform closely to our intuition about the relations between these semantics.

Our approach is summarized in Figure 3.1. Note that the specialization process of off-line PPE can be systematically and correctly derived from its on-line counterpart, using the information collected in the facet-analysis phase.

3.2 Core Semantics

We begin the discussion of the semantic specification of PPE by presenting a core semantics. The subject language is a first-order functional language. Figure 3.2 defines its syntactic domains. The meaning of a program is the meaning of function f_1. We assume all functions (and primitive operations) have the same arity.

The core semantics is defined in Figure 3.3. It is used as a basis for all the other semantic specifications defined later, and it factors out the common components of those semantic specifications. This semantics is composed of two valuation functions: $\overline{\mathcal{E}}$ and $\overline{\mathcal{A}}$. Briefly, $\overline{\mathcal{E}}$ defines the standard/abstract semantics (called the *local* semantics) for the language constructs, while $\overline{\mathcal{A}}$ defines a process which collects information globally (called the *global* semantics). The structure of the core semantics is similar to that used in [Hudak and Young, 1991] for defining collecting interpretation. A similar structure is also used in [Sestoft, 1985] to define a binding-time analysis.

The core semantics is defined by semantic rules. It uses some uninterpreted domain names and combinator names. A specific semantics is defined by providing an interpretation to these domains and combinators. As a result, relation between two instantiated semantics can simply be defined by relating their domains and their combinators. Indeed, all three semantic specifications presented in this chapter are defined from the core semantics displayed in Figure 3.3. Also, their correctness are proven using the relations defined between their domains and combinators, as is depicted in Figure 3.4.

3.3 Standard and Instrumented Semantics

3.3.1 The Semantics Specifications

In Figure 3.5, we instantiate the core semantics to define the standard semantics of our language. As is customary, we will omit summand projections and injections.

$$
\begin{array}{rcll}
c & \in & \textbf{Const} & \text{Constants} \\
x & \in & \textbf{Var} & \text{Variables} \\
p & \in & \textbf{Po} & \text{Primitive Operators} \\
f & \in & \textbf{Fn} & \text{Function Names} \\
e & \in & \textbf{Exp} & \text{Expressions} \\
e & ::= & c \mid x \mid p\,(e_1, \ldots, e_n) \mid f\,(e_1, \ldots, e_n) \mid \mathit{if}\ e_1\ e_2\ e_3 & \\
\textbf{Prog} & ::= & \{f_i(x_1, \ldots, x_n) = e_i\} \quad (f_1 \text{ is the main function}) &
\end{array}
$$

Figure 3.2: Syntactic Domains of the Subject Language

1. $\overline{\mathcal{E}} : \textbf{Exp} \rightarrow \mathit{ECont} \quad \mathit{where}\ \mathit{ECont} = \overline{\mathit{Env}} \rightarrow \mathit{Result}_{\overline{\mathcal{E}}}$

$$
\begin{array}{l}
\overline{\mathcal{E}}[\![c]\!] = \mathit{Const}_{\overline{\mathcal{E}}}\,[\![c]\!] \\
\overline{\mathcal{E}}[\![x]\!] = \mathit{VarLookup}_{\overline{\mathcal{E}}}\,[\![x]\!] \\
\overline{\mathcal{E}}[\![p(e_1, \ldots, e_n)]\!] = \mathit{PrimOp}_{\overline{\mathcal{E}}}\,[\![p]\!](\overline{\mathcal{E}}[\![e_1]\!], \ldots, \overline{\mathcal{E}}[\![e_n]\!]) \\
\overline{\mathcal{E}}[\![\mathit{if}\ e_1\ e_2\ e_3]\!] = \mathit{Cond}_{\overline{\mathcal{E}}}\,(\overline{\mathcal{E}}[\![e_1]\!],\ \overline{\mathcal{E}}[\![e_2]\!],\ \overline{\mathcal{E}}[\![e_3]\!]) \\
\overline{\mathcal{E}}[\![f(e_1, \ldots, e_n)]\!] = \mathit{App}_{\overline{\mathcal{E}}}\,[\![f]\!](\overline{\mathcal{E}}[\![e_1]\!], \ldots, \overline{\mathcal{E}}[\![e_n]\!])
\end{array}
$$

$\mathit{where}\ \mathit{Const}_{\overline{\mathcal{E}}} : \textbf{Const} \rightarrow \mathit{ECont}$

$\mathit{VarLookup}_{\overline{\mathcal{E}}} : \textbf{Var} \rightarrow \mathit{ECont}$

$\mathit{PrimOp}_{\overline{\mathcal{E}}} : \textbf{Po} \rightarrow \mathit{ECont}^n \rightarrow \mathit{ECont}$

$\mathit{Cond}_{\overline{\mathcal{E}}} : \mathit{ECont}^3 \rightarrow \mathit{ECont}$

$\mathit{App}_{\overline{\mathcal{E}}} : \textbf{Fn} \rightarrow \mathit{ECont}^n \rightarrow \mathit{ECont}$

2. $\overline{\mathcal{A}} : \textbf{Exp} \rightarrow \mathit{ACont} \quad \mathit{where}\ \mathit{ACont} = \overline{\mathit{Env}} \rightarrow \mathit{Result}_{\overline{\mathcal{A}}}$

$$
\begin{array}{l}
\overline{\mathcal{A}}[\![c]\!] = \mathit{Const}_{\overline{\mathcal{A}}}\,[\![c]\!] \\
\overline{\mathcal{A}}[\![x]\!] = \mathit{VarLookup}_{\overline{\mathcal{A}}}\,[\![x]\!] \\
\overline{\mathcal{A}}[\![p(e_1, \ldots, e_n)]\!] = \mathit{PrimOp}_{\overline{\mathcal{A}}}\,[\![p]\!]\,(\overline{\mathcal{A}}[\![e_1]\!], \ldots,\ \overline{\mathcal{A}}[\![e_n]\!]) \\
\overline{\mathcal{A}}[\![\mathit{if}\ e_1\ e_2\ e_3]\!] = \mathit{Cond}_{\overline{\mathcal{A}}}\,(\overline{\mathcal{A}}[\![e_1]\!],\ \overline{\mathcal{A}}[\![e_2]\!],\ \overline{\mathcal{A}}[\![e_3]\!])\ (\overline{\mathcal{E}}[\![e_1]\!]) \\
\overline{\mathcal{A}}[\![f(e_1, \ldots, e_n)]\!] = \mathit{App}_{\overline{\mathcal{A}}}\,[\![f]\!]\ (\overline{\mathcal{A}}[\![e_1]\!], \ldots,\ \overline{\mathcal{A}}[\![e_n]\!])\ (\overline{\mathcal{E}}[\![e_1]\!], \ldots,\ \overline{\mathcal{E}}[\![e_n]\!])
\end{array}
$$

$\mathit{where}\ \mathit{Const}_{\overline{\mathcal{A}}} : \textbf{Const} \rightarrow \mathit{ACont}$

$\mathit{VarLookup}_{\overline{\mathcal{A}}} : \textbf{Var} \rightarrow \mathit{ACont}$

$\mathit{PrimOp}_{\overline{\mathcal{A}}} : \textbf{Po} \rightarrow \mathit{ACont}^n \rightarrow \mathit{ACont}$

$\mathit{Cond}_{\overline{\mathcal{A}}} : \mathit{ACont}^3 \rightarrow \mathit{ECont} \rightarrow \mathit{ACont}$

$\mathit{App}_{\overline{\mathcal{A}}} : \textbf{Fn} \rightarrow \mathit{ACont}^n \rightarrow \mathit{ECont}^n \rightarrow \mathit{ACont}$

Figure 3.3: Core Semantics

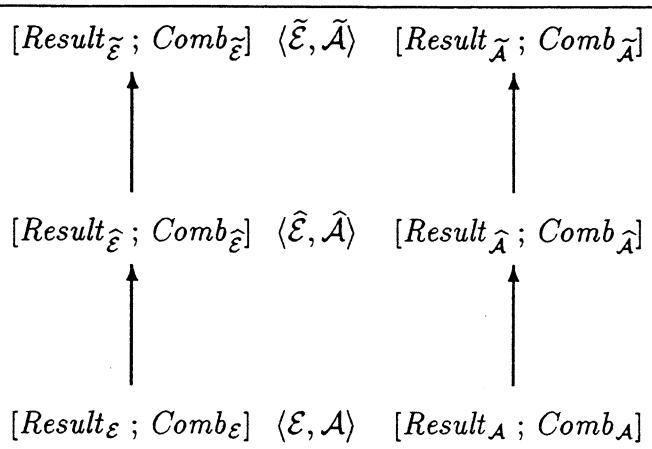

Figure 3.4: Relations between three levels of evaluation

Result$_{\overline{\mathcal{E}}}$ and *Result*$_{\overline{\mathcal{A}}}$ are the result domains used by semantic functions $\overline{\mathcal{E}}$ and $\overline{\mathcal{A}}$ respectively. *Comb*$_{\overline{\mathcal{E}}}$ and *Comb*$_{\overline{\mathcal{A}}}$ are their respective set of combinators.

- Semantic Domains

  $v \in \mathit{Result}_{\mathcal{E}} = \mathbf{Values} = \mathbf{Int} + \mathbf{Bool}$

  $\rho \in \mathit{VarEnv} = \mathbf{Var} \rightarrow \mathbf{Values}$

  $\phi \in \mathit{FunEnv} = \mathbf{Fn} \rightarrow \mathbf{Values}^n \rightarrow \mathbf{Values}$

  $\mathit{Env} = \mathit{VarEnv} \times \mathit{FunEnv}$

- Valuation Functions

  $\mathcal{E}_{\mathit{Prog}} : \mathbf{Prog} \rightarrow \mathbf{Values}^n \rightarrow \mathbf{Values}$

  $\mathcal{E}_{\mathit{Prog}}\ [\![\{\ f_i(x_1, \ldots, x_n) = e_i\}]\!] \langle v_1, \ldots, v_n \rangle = \phi\ [\![f_1]\!]\ (v_1, \ldots, v_n)$

  $\quad \mathit{whererec}\ \phi = \bot[\mathit{strict}\ \{\lambda(v_1, \ldots, v_n)\ .\ \mathcal{E}[\![e_i]\!]\ ((\bot[v_k/x_k]), \phi)\}/f_i]$

  $\mathcal{E} = \overline{\mathcal{E}}$

- Combinator Definitions

  $\mathit{Const}_{\mathcal{E}}\ [\![c]\!] = \lambda(\rho, \phi)\ .\ \mathcal{K}\ [\![c]\!]$

  $\mathit{VarLookup}_{\mathcal{E}}\ [\![x]\!] = \lambda(\rho, \phi)\ .\ \rho[\![x]\!]$

  $\mathit{PrimOp}_{\mathcal{E}}\ [\![p]\!]\ (k_1, \ldots, k_n) = \lambda(\rho, \phi)\ .\ \mathcal{K}_P[\![p]\!]\ (k_1(\rho, \phi), \ldots, k_n(\rho, \phi))$

  $\mathit{Cond}_{\mathcal{E}}\ (k_1,\ k_2,\ k_3) = \lambda(\rho, \phi)\ .\ k_1(\rho, \phi) \rightarrow k_2(\rho, \phi),\ k_3(\rho, \phi)$

  $\mathit{App}_{\mathcal{E}}\ [\![f]\!]\ (k_1, \ldots, k_n) = \lambda(\rho, \phi)\ .\ \phi[\![f]\!]\ (k_1(\rho, \phi), \ldots, k_n(\rho, \phi))$

Figure 3.5: Standard Semantics

Only the interpretation of the valuation function $\overline{\mathcal{E}}$ is provided since the definition of standard semantics does not require collecting information globally. For a function f, "*strict* f" is a function just like f except that it is strict in all its arguments.

In order to investigate the relationship between the standard semantics and the PPE semantics, the former is enriched to capture information about function applications. This enhanced semantics, called *instrumented* semantics, collects all function calls performed during the standard execution of a program. Function calls are recorded in a *cache*, which maps a function name to a set of *standard signatures.*[2] A standard signature consists of the argument values of a function application. This is depicted in figures 3.6 and 3.7.

The continuity property of the function $\mathcal{E}$ is well-known, for the function is the same as conventional standard semantic function for a first-order applicative language. Since the function $\mathcal{A}$ describes a kind of collecting interpretation (over function calls, instead of all expressions), its continuity can be proven along the same argument described in [Hudak and Young, 1991]. This involves rewriting the specification of $\mathcal{A}$ into a functional, and showing the continuity of that functional.[3] The detail is omitted in this thesis.

3.3.2 Correctness of Instrumentation

Because the local semantics is exactly identical to the standard semantics, we only need to show that the instrumentation part of the instrumented semantics is correct. That is, the instrumented semantics captures (in the cache) all the calls performed during standard evaluation. Since the language we consider is strict, only those

[2]Notice that powerset, instead of powerdomain, is used to model the content of the cache. This avoids some technical complication incurred in the correctness proof, as discussed in [Hudak and Young, 1991].

[3]In fact, the proof required here, as well as those for the continuity property of both on-line and off-line PPE, is simpler than the one described in [Hudak and Young, 1991]. This is because (1) the subject language is first-order applicative, (2) information is collected only at each call site, instead of all expressions, and (3) treatment of function application is uniform (*i.e.*, collecting the result of processed arguments), independent of the function definition.

- Semantic Domains

  $v \in \mathit{Result}_{\mathcal{E}} = \mathbf{Values} = \textit{as in Figure 3.5}$
  $\rho \in \mathit{VarEnv} = \textit{as in Figure 3.5}$
  $\phi \in \mathit{FunEnv} = \textit{as in Figure 3.5}$
  $\sigma \in \mathit{Result}_{\mathcal{A}} = \mathbf{Cache}_{\mathcal{A}} = \mathbf{Fn} \rightarrow \mathcal{P}(\mathbf{Values}^n)$

- Valuation Functions

  $\mathcal{E}_{Prog} : \mathbf{Prog} \rightarrow \mathbf{Values}^n \rightarrow \mathbf{Cache}_{\mathcal{A}}$
  $\mathcal{E}_{Prog}\ [\![\{\ f_i(x_1,\ldots,x_n) = e_i\}]\!]\langle v_1,\ldots,v_n\rangle = h(\bot[\{\langle v_1,\ldots,v_n\rangle\}/f_1])$
  $\quad \textit{whererec}\quad h(\sigma) = \sigma \sqcup$
  $\qquad h(\bigsqcup\{\mathcal{A}\ [\![e_i]\!]\ (\bot[v_k/x_k])\phi \mid \langle v_1,\ldots,v_n\rangle \in \sigma[\![f_i]\!],$
  $\qquad\qquad [\![f_i]\!] \in \mathit{Dom}(\sigma)\})$
  $\qquad \phi = \bot[\mathit{strict}\ (\lambda(v_1,\ldots,v_n)\ .\ \mathcal{E}\ [\![e_i]\!]\ (\bot[v_k/x_k])\ \phi)/f_i]$
  $\mathcal{E} = \overline{\mathcal{E}}$
  $\mathcal{A} = \overline{\mathcal{A}}$

Figure 3.6: Instrumented Semantics — Domains and Main functions

standard signatures that represent function calls with non-bottom argument values are collected in the cache. We shall refer to these function calls as *non-trivial* calls. The following lemma describes the behavior of the instrumented semantic function $\mathcal{A}$. It is used to show the correctness of instrumentation.

Lemma 3.1 *Given a program P. Let ϕ be the function environment for P defined by the instrumented semantics. If the standard evaluation of P with input $\langle v_1,\ldots,v_n\rangle$ terminates, and σ is the cache computed for P by $\mathcal{A}$, then*

1. *For any expression e in P, if a non-trivial function call occurring in e is performed when e is evaluated, then $\mathcal{A}$ records the call in the cache.*

2. *For any function definition in P of the form*

$$f_i(x_1,\ldots,x_n) = \cdots\ f_j(e'_1,\ldots,e'_n)\ \cdots$$

Let $\langle v'_1,\ldots,v'_n\rangle \in \sigma[\![f_i]\!]$. If evaluating f_i with argument $\langle v'_1,\ldots,v'_n\rangle$ results in a call to f_j with $\langle v''_1,\ldots,v''_n\rangle$, where $v''_l = \mathcal{E}[\![e'_l]\!](\bot[v'_k/x_k],\phi)\ \forall l \in \{1,\ldots,n\}$,

- Combinator Definitions

$Const_{\mathcal{E}}\ [\![c]\!] = as\ in\ Figure\ 3.5$
$VarLookup_{\mathcal{E}}\ [\![x]\!] = as\ in\ Figure\ 3.5$
$PrimOp_{\mathcal{E}}\ [\![p]\!]\ (k_1, \ldots, k_n) = as\ in\ Figure\ 3.5$
$Cond_{\mathcal{E}}\ (k_1,\ k_2,\ k_3) = as\ in\ Figure\ 3.5$
$App_{\mathcal{E}}\ [\![f]\!]\ (k_1, \ldots, k_n) = as\ in\ Figure\ 3.5$

$$Const_{\mathcal{A}}\ [\![c]\!] = \lambda(\rho, \phi)\ .\ (\lambda f\ .\ \{\})$$
$$VarLookup_{\mathcal{A}}\ [\![x]\!] = \lambda(\rho, \phi)\ .\ (\lambda f\ .\ \{\})$$
$$PrimOp_{\mathcal{A}}\ [\![p]\!]\ (a_1, \ldots, a_n) = \lambda(\rho, \phi)\ .\ \bigsqcup_{i=1}^{n} a_i(\rho, \phi)$$
$$Cond_{\mathcal{A}}\ (a_1,\ a_2,\ a_3)\ k_1 = \lambda(\rho, \phi)\ .\ a_1(\rho, \phi) \sqcup (k_1(\rho, \phi) \rightarrow a_2(\rho, \phi),\ a_3(\rho, \phi))$$
$$App_{\mathcal{A}}\ [\![f]\!]\ (a_1, \ldots,\ a_n)\ (k_1, \ldots,\ k_n) =$$
$$\lambda(\rho, \phi)\ .\ \bigsqcup_{i=1}^{n} a_i(\rho, \phi)) \sqcup\ ((\exists i \in \{1, \ldots, n\}\ s.t. v_i = \bot) \rightarrow (\lambda f\ .\ \{\}),\ \bot[\{\langle v_1, \ldots, v_n\rangle\}/f])$$
$$where\ v_i = k_i(\rho, \phi)\quad \forall\, i \in \{1, \ldots, n\}$$

Figure 3.7: Instrumented Semantics — Global semantic function

then $\langle v_1'', \ldots, v_n''\rangle \in \sigma[\![f_j]\!]$, *provided* $v_l'' \neq \bot, \forall l \in \{1, \ldots, n\}$.

Proof (Sketch):

1. We want to show that the predicate "if a non-trivial function call occurring in e is performed when evaluating e, then the call is recorded in the cache produced by $\mathcal{A}$" is true. The proof is done by structural induction over e.

2. The second part of the lemma is shown by examining local function h in function $\mathcal{E}_{Prog}$. If $\langle v_1', \ldots, v_n'\rangle \in \sigma[\![f_i]\!]$, then $\mathcal{A}$ will be called to collect non-trivial calls occurring in the evaluation of f_i with $\bot[v_k'/x_k]$ as its variable environment. Using the first result of this lemma we know that $\langle v_1'', \ldots, v_n''\rangle \in \sigma[\![f_j]\!]$, provided $v_l'' \neq \bot, \forall l \in \{1, \ldots, n\}$. □

Theorem 3.1 (Correctness of Instrumentation) *Given a program P, let P be evaluated with input $\langle v_1, \ldots, v_n\rangle$. For any user-defined function f in P, if f is*

called with non-bottom argument $\langle v'_1, \ldots, v'_n \rangle$ *during the standard evaluation, then* $\langle v'_1, \ldots, v'_n \rangle \in \sigma[\![f]\!]$.

Proof : From Lemma 3.1, and noticing that, since none of the initial input should be bottom, the initial call to f_1 with argument $\langle v_1, \ldots, v_n \rangle$ is captured in the cache (by the definition of $\mathcal{E}_{Prog}$). □

3.4 On-Line PPE

In this section, we instantiate the core semantics to obtain an on-line PPE semantics. Using logical relation, we then prove the correctness of this on-line PPE semantics.

3.4.1 The Semantics Specification

Figures 3.8, 3.9 and 3.10 display the on-line PPE semantics. The semantics aims at partially evaluating a program with respect to a set of static properties (including constants). It yields a residual program consisting of the specialized functions created at partial-evaluation time. As was described in Section 2.3.4, we assume that the partial-evaluation facet always exists during partial evaluation — 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. For brevity, We write $\top_{\widehat{\mathcal{SD}}}$ to represent the maximum value of any summand of $\widehat{\mathcal{SD}}$.

Partial evaluation of an expression always produces a residual expression. Besides using $[\![\ ]\!]$ to denote an expression, we also use it as an operator that constructs expressions. This operation is assumed to be strict in all its arguments (*i.e.*, the subexpressions). To this end, domain **Exp** containing all expressions forms a flat domain.

The semantics consists of three valuation functions: $\widehat{\mathcal{E}}_{Prog}$, $\widehat{\mathcal{E}}$ and $\widehat{\mathcal{A}}$. Function $\widehat{\mathcal{E}}_{Prog}$ begins the partial-evaluation process by calling function $\widehat{\mathcal{A}}$ to collect information from

function calls encountered during partial evaluation. At the end of the process, it calls function *MkProg* to construct the residual program.

- Semantic Domains

$\hat{\delta} \in \widehat{\mathcal{SD}} = \sum_{j=1}^{s} \hat{\mathcal{D}}_j$ *where* $\hat{\mathcal{D}}_j = (\widehat{\mathbf{D}}_j^1 \otimes \cdots \otimes \widehat{\mathbf{D}}_j^m)$
and s is the number of basic domains

$\hat{v} \in \mathit{Result}_{\hat{\mathcal{E}}} = \mathbf{Exp} \times \widehat{\mathcal{SD}}$

$\hat{\rho} \in \widehat{\mathit{VarEnv}} = \mathbf{Var} \to \mathit{Result}_{\hat{\mathcal{E}}}$

$\widehat{\mathit{Env}} = \widehat{\mathit{VarEnv}} \times \widehat{\mathit{FunEnv}}$

$\hat{\phi} \in \widehat{\mathit{FunEnv}} = \mathbf{Fn} \to \mathit{Result}_{\hat{\mathcal{E}}}^n \to \mathit{Result}_{\hat{\mathcal{E}}}$

$\hat{\sigma} \in \mathit{Result}_{\hat{\mathcal{A}}} = \mathbf{Cache}_{\hat{\mathcal{A}}} = \mathbf{Fn} \to \mathcal{P}(\mathbf{Transf} \times \mathit{Result}_{\hat{\mathcal{E}}}^n)$

- Valuation Functions

$\hat{\mathcal{E}}_{\mathit{Prog}} : \mathbf{Prog} \to \mathit{Result}_{\hat{\mathcal{E}}}^n \to \mathbf{Prog}_{\perp}$

$\hat{\mathcal{E}}_{\mathit{Prog}} [\![\{f_i(x_1, \ldots, x_n) = e_i\}]\!] (\hat{v}_1, \ldots, \hat{v}_n) =$
$\mathit{MkProg}\ (\hat{h}(\perp[\{\langle \mathbf{s}, \hat{v}_1, \ldots, \hat{v}_n\rangle\}/f_1]))\hat{\phi}$
$\mathit{whererec}\ \hat{h}(\hat{\sigma}) = \hat{\sigma} \sqcup \hat{h}(\bigsqcup\{\hat{\mathcal{A}}\ [\![e_i]\!]\ (\perp[\hat{v}'_k/x_k], \hat{\phi}) \mid \langle -, \hat{v}'_1, \ldots, \hat{v}'_n\rangle \in \hat{\sigma}[\![f_i]\!],$
$[\![f_i]\!] \in \mathit{Dom}(\hat{\sigma})\})$
$\hat{\phi} = \perp[\mathit{strict}\ (\lambda(\hat{v}_1, \ldots, \hat{v}_n)\ .\ \hat{\mathcal{E}}\ [\![e_i]\!]\ (\perp[\hat{v}_k/x_k], \hat{\phi}))/f_i]$

$\hat{\mathcal{E}} = \overline{\mathcal{E}}$

$\hat{\mathcal{A}} = \overline{\mathcal{A}}$

- *MkProg* Definition

$\mathit{MkProg}\ \hat{\sigma}\ \hat{\phi} = \{\ f_i^{sp}(x_1, \ldots, x_k) = \hat{v}{\downarrow}1 \mid \langle \mathbf{s}, \hat{v}_1, \ldots, \hat{v}_n\rangle \in \hat{\sigma}[\![f_i]\!],$
$[\![f_i]\!] \in \mathit{Dom}(\hat{\sigma})\}$
$\mathit{where}\ f_i^{sp} = \mathit{SpName}([\![f_i]\!], \hat{v}_1, \ldots, \hat{v}_n)$
$\hat{v} = \hat{\mathcal{E}}\ [\![e_i]\!]\ (\perp[\hat{v}_k/x_k], \hat{\phi})$
$\langle x_1, \ldots, x_k\rangle = \mathit{ResidPars}\ ([\![f_i]\!], \hat{v}_1{\downarrow}1, \ldots, \hat{v}_n{\downarrow}1)$

Figure 3.8: On-Line PPE — Domains and Main functions

Function $\hat{\mathcal{E}}$ defines partial evaluation of an expression. It yields a value $\hat{v} \in \mathit{Result}_{\hat{\mathcal{E}}} = \mathbf{Exp} \times \widehat{\mathcal{SD}}$, where the first component ($\hat{v}{\downarrow}1$) is a residual expression and the second component ($\hat{v}{\downarrow}2$) is a product-of-facet value. Domain $\mathit{Result}_{\hat{\mathcal{E}}}$ is partially ordered component-wise.

- Local Combinator Definitions

$Const_{\widehat{\mathcal{E}}}\ [\![c]\!] = \lambda(\hat{\rho},\hat{\phi})\ .\ \widehat{\mathcal{K}}\ [\![c]\!]$

$VarLookup_{\widehat{\mathcal{E}}}\ [\![x]\!] = \lambda(\hat{\rho},\hat{\phi})\ .\ \hat{\rho}\ [\![x]\!]$

$PrimOp_{\widehat{\mathcal{E}}}\ [\![p]\!]\ (\hat{k}_1,\ldots,\hat{k}_n) = \lambda(\hat{\rho},\hat{\phi})\ .\ \widehat{\mathcal{K}}_P\ [\![p]\!]\ (\hat{k}_1(\hat{\rho},\hat{\phi}),\ldots,\ \hat{k}_n(\hat{\rho},\hat{\phi}))$

$Cond_{\widehat{\mathcal{E}}}\ (\hat{k}_1,\hat{k}_2,\hat{k}_3) = \lambda(\hat{\rho},\hat{\phi})\ .\ (\hat{v}_1{\downarrow}1 \in \mathbf{Const}) \rightarrow ((\mathcal{K}(\hat{v}_1{\downarrow}1)) \rightarrow \hat{v}_2, \hat{v}_3),$
$\langle [\![if\ \hat{v}_1{\downarrow}1\ \hat{v}_2{\downarrow}1\ \hat{v}_3{\downarrow}1]\!], \hat{v}_2{\downarrow}2 \sqcup \hat{v}_3{\downarrow}2\rangle$

where $\hat{v}_i = \hat{k}_i(\hat{\rho},\hat{\phi}) \quad \forall i \in \{1,\ 2,\ 3\}$

$App_{\widehat{\mathcal{E}}}\ [\![f]\!]\ (\hat{k}_1,\ldots,\hat{k}_n) = \lambda(\hat{\rho},\hat{\phi})\ .\ (Ft\ [\![f]\!]){\downarrow}1\ (\widehat{bt}(\hat{v}_1),\ldots,\widehat{bt}(\hat{v}_n)) = \mathtt{u}$
$\rightarrow\ \hat{\phi}\ [\![f]\!]\ (\hat{v}_1,\ldots,\hat{v}_n),$
$\langle [\![f_{sp}(e''_1,\ldots,e''_k)]\!], \hat{\delta}'\rangle$

where $\hat{v}_i = \hat{k}_i(\hat{\rho},\hat{\phi}) \quad \forall i \in \{1,\ldots,n\}$
$f_{sp} = SpName([\![f]\!], \hat{v}'_1,\ldots,\hat{v}'_n)$
$\langle e''_1,\ldots,e''_k\rangle = ResidArgs\ ([\![f]\!],\ \langle b_1,\ldots,b_n\rangle,\ \langle \hat{v}_1{\downarrow}1,\ldots,\hat{v}_n{\downarrow}1\rangle)$
$\langle e', \hat{\delta}'\rangle = \hat{\phi}\ [\![f]\!]\ (\hat{v}'_1,\ldots,\hat{v}'_n)$
$\langle \hat{v}'_1,\ldots,\hat{v}'_n\rangle = SpPat\ ([\![f]\!], \langle \hat{v}_1,\ldots,\hat{v}_n\rangle,\ \langle b_1,\ldots,b_n\rangle)$
$\langle b_1,\ldots,b_n\rangle = (Ft\ [\![f]\!]){\downarrow}2\ (\widehat{bt}(\hat{v}_1),\ldots,\widehat{bt}(\hat{v}_n))$

- Primitive Functions

$\widehat{\mathcal{K}}\ :\ \mathbf{Const} \rightarrow Result_{\widehat{\mathcal{E}}}$
$\widehat{\mathcal{K}}\ [\![c]\!] = \langle [\![c]\!], \langle \hat{\alpha}_{\widehat{\mathcal{D}}^1}(d),\ \ldots, \hat{\alpha}_{\widehat{\mathcal{D}}^m}(d)\rangle\rangle \quad where\ d = \mathcal{K}\ [\![c]\!] \in \mathbf{D}$

$\widehat{\mathcal{K}}\ :\ \mathbf{Po} \rightarrow Result^n_{\widehat{\mathcal{E}}} \rightarrow Result_{\widehat{\mathcal{E}}}$
$\widehat{\mathcal{K}}_P\ [\![p^c]\!]\ (\langle e'_1,\hat{\delta}_1\rangle,\ldots,\langle e'_n,\hat{\delta}_n\rangle) =$
$(\hat{\delta} = \bot_{\widehat{\mathcal{D}}}) \rightarrow \langle \bot_{Exp}, \bot_{\widehat{\mathcal{SD}}}\rangle,$
$(\hat{\delta}^1 \in \mathbf{Const}) \rightarrow\ \langle \hat{\delta}^1, \langle \hat{\alpha}_{\widehat{\mathcal{D}}^1}(d),\ldots,\ \hat{\alpha}_{\widehat{\mathcal{D}}^m}(d)\rangle\rangle,\ \langle e', \hat{\delta}\rangle$

where $p^c\ :\ \mathbf{D}^n \rightarrow \mathbf{D}$
$\hat{\delta} = \hat{\omega}_{p^c}(\hat{\delta}_1,\ldots,\hat{\delta}_n)$
$e' = [\![p^c(e'_1,\ldots,e'_n)]\!]$
$d = \mathcal{K}(\hat{\delta}^1)$

$\widehat{\mathcal{K}}_P\ [\![p^o]\!]\ (\langle e'_1,\hat{\delta}_1\rangle,\ldots,\langle e'_n,\hat{\delta}_n\rangle) =$
$(\hat{d} = \bot_{\widehat{Values}}) \rightarrow \langle \bot_{Exp}, \bot_{\widehat{\mathcal{SD}}}\rangle,$
$\hat{d} \in \mathbf{Const} \rightarrow\ \langle \hat{d}, \langle \hat{\alpha}_{\widehat{\mathcal{D}'}^1}(d),\ldots,\hat{\alpha}_{\widehat{\mathcal{D}'}^m}(d)\rangle\rangle,\ \langle e', \langle \top_{\widehat{\mathcal{D}'}^1},\ \ldots,\ \top_{\widehat{\mathcal{D}'}^m}\rangle\rangle$

where $p^o\ :\ \mathbf{D}^n \rightarrow \mathbf{D}'$
$\hat{d} = \hat{\omega}_{p^o}(\hat{\delta}_1,\ldots,\hat{\delta}_n)$
$e' = [\![p^o(e'_1,\ldots,e'_n)]\!]$
$d = \mathcal{K}(\hat{d})$

Figure 3.9: On-Line PPE — Local Semantics

- Global Combinator Definitions

$$Const_{\widehat{\mathcal{A}}}\ [\![c]\!] \;=\; \lambda(\hat{\rho},\hat{\phi})\ .\ (\lambda f\ .\ \{\})$$

$$VarLookup_{\widehat{\mathcal{A}}}\ [\![x]\!] \;=\; \lambda(\hat{\rho},\hat{\phi})\ .\ (\lambda f\ .\ \{\})$$

$$PrimOp_{\widehat{\mathcal{A}}}\ [\![p]\!]\ (\hat{a}_1,\ldots,\hat{a}_n) \;=\; \lambda(\hat{\rho},\hat{\phi})\ .\ \bigsqcup_{i=1}^{n} \hat{a}_i(\hat{\rho},\hat{\phi})$$

$$Cond_{\widehat{\mathcal{A}}}\ (\hat{a}_1,\hat{a}_2,\hat{a}_3)\ \hat{k}_1 \;=\lambda(\hat{\rho},\hat{\phi})\ .\ \ \hat{a}_1(\hat{\rho},\hat{\phi}) \sqcup \hat{\delta}_1^1 \in \mathbf{Const} \rightarrow\ (\mathcal{K}(\hat{\delta}_1^1) \rightarrow \hat{a}_2(\hat{\rho},\hat{\phi}), \hat{a}_3(\hat{\rho},\hat{\phi})),\ \hat{a}_2(\hat{\rho},\hat{\phi}) \sqcup \hat{a}_3(\hat{\rho},\hat{\phi})$$

$$\textit{where}\ \langle e_1, \langle \hat{\delta}_1^1,\ldots,\hat{\delta}_1^m\rangle\rangle \;=\; \hat{k}_1(\hat{\rho},\hat{\phi})$$

$$App_{\widehat{\mathcal{A}}}\ [\![f]\!]\ (\hat{a}_1,\ldots,\ \hat{a}_n)\ (\hat{k}_1,\ldots,\ \hat{k}_n) \;=\; \lambda(\hat{\rho},\hat{\phi})\ .\ (\bigsqcup_{i=1}^{n} \hat{a}_i(\hat{\rho},\hat{\phi})) \sqcup \hat{\sigma}$$

$$\textit{where}\ \hat{\sigma} \;=\; ((Ft[\![f]\!])\!\downarrow\!1(\widehat{bt}(\hat{v}_1),\ldots,\widehat{bt}(\hat{v}_n)) = \mathtt{u}) \rightarrow \bot[\{\langle \mathtt{u},\hat{v}_1,\ldots,\hat{v}_n\rangle\}/f],\ \bot[\{\langle \mathtt{s},\hat{v}_1',\ldots,\hat{v}_n'\rangle\}/f]$$

$$\langle \hat{v}_1',\ldots,\hat{v}_n'\rangle \;=\; SpPat\ ([\![f]\!], \langle \hat{v}_1,\ldots,\hat{v}_n\rangle,\ \langle b_1,\ldots,b_n\rangle)$$

$$\langle b_1,\ldots,b_n\rangle \;=\; (Ft\ [\![f]\!])\!\downarrow\!2\ (\widehat{bt}(\hat{v}_1),\ldots,\widehat{bt}(\hat{v}_n))$$

$$\hat{v}_i \;=\; \hat{k}_i(\hat{\rho},\hat{\phi}) \quad \forall i \in \{1,\ldots,n\}$$

Figure 3.10: On-Line PPE — Global semantic function

3.4.2 Treatment of Function Calls

One of the central issues in partial evaluation of functional programs is the treatment of function calls. Basically, there are two kinds of transformation performed in partially evaluating a function call: *unfold* and *specialization*. Each transformation has a major pitfall which may cause non-termination of the partial-evaluation process. These pitfalls are infinite unfolding and infinite specialization.

This section first presents the treatment of function calls. Then, we use the notion of *filter* to capture various call treatments [Consel, 1989].

Call Unfolding

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 arguments.

As an example, consider the following function[4], which appends n numbers (from m to m+n) to a list l.[5]

```
fun appendn (n m l) =
    if (n < 0) then l else m :: (appendn (n-1,m+1,l))
```

If the function call appendn(2,4,v), where variable v is dynamic, is unfolded, the resulting expression is 4::5::6::v . In this case, unfolding is safe (*i.e.*, it will terminate) because the induction variable n is static. Although safe, unfolding may cause computations to be duplicated. This happens when a parameter occurs more than once and its corresponding argument is dynamic. However, this can easily be avoided by a preliminary analysis as described in [Bondorf and Danvy, 1991].

Not all function calls can be unfolded. Consider unfolding appendn(u,v,[3,4]), where u and v are dynamic. In this context, the recursion of function appendn is under dynamic control. Therefore, systematic unfolding of calls to this function will cause non-termination of partial evaluation.

When unfolding cannot be performed, the function call has to be suspended.

Call Suspension

When a function call is suspended, a specialized version of the function is created with respect to the static arguments. A new function call (called the *residual call*) consisting of the name of the specialized function and the dynamic arguments is substituted for the original function call. Consider suspending the function call appendn(u,v,[3,4]) with both u and v being dynamic. The resulting specialized function is

```
fun appendn-1(n,m) =
    if (n < 0) then [3,4] else m :: (appendn-1(n-1,m+1))
```

[4]Although contrived, this example illustrates many aspects of the treatment of function calls.

[5]All program code are written in a simplified format of Standard ML [Milner *et al.*, 1990]. As such, operator :: represents the "consing" of an element to form a list, and [...] forms a list, with the empty list represented by [].

and the residual call is `appendn-1(u,v)`. Since a function is specialized with respect to static arguments, calls with different static arguments produce different specialized functions (thus attaining *polyvariant specialization*). Hence, when suspending a function call, the partial evaluator first determines whether a specialized version of the function has already been created via a previously suspended call with the same static arguments. If so, it does not create a new specialized function; rather, it replaces the original call by a residual call to this specialized function. This is illustrated in the above specialized function `appendn-1`, where the recursive call also refers to `appendn-1` since this call, when suspended, contains the same static arguments as the previously suspended call.

It is not always safe to specialize a function with respect to all the static arguments of the call; some arguments may cause infinite specialization [Consel, 1989]. Consider a systematic suspension of all calls to `appendn`, with the first call being `appendn(u,2,[3,4])` where variable `u` is dynamic. Suspending this first call causes a specialized version of `appendn` to be created, using the pair of static values $\langle 2, [3,4] \rangle$. Since the induction variable is dynamic, both branches of the conditional in `appendn` are partially evaluated. Consequently, suspending the recursive call to `appendn` causes a new specialized version of `appendn` to be created, using the pair $\langle 3, [3,4] \rangle$. Since the termination condition is dynamic, each recursive call to `appendn` will yield a new specialized function and this process will not terminate. To prevent infinite specialization, some static arguments should not be propagated.

Filters

Exactly how a function call is to be treated can be determined by the user, or automatically by some termination analysis (*e.g.*, [Sestoft, 1988]). To capture this piece of decision making, we introduce the notion of *filters*.

Using filter to specify the treatment of a function call has been developed for the conventional partial evaluator Schism [Consel, 1988,Consel, 1990b]. In this scheme, each user-defined function in a subject program can be associated with a filter speci-

fication.

A filter specifies how to transform a function call (unfolding/suspension) and how to specialize a function, when call suspension occurs. A filter consists of a pair of *strict* and *continuous* functions. The first component of the filter specifies whether a call to this function should be unfolded or suspended. It has the functionality ($\widetilde{\textbf{Values}}^n \rightarrow \textbf{Transf}$) where $\widetilde{\textbf{Values}}$ is the binding-time domain (see Definition 2.3) and domain **Transf** contains two values: `u` and `s`, which stand for unfolding and specializing respectively. This function is invoked with the binding-time values of the arguments of a call. If it returns `u`, the call is unfolded. Otherwise, it returns `s`; the call is then suspended and the function is specialized.

Domain **Transf** is ordered as follows: $\texttt{u} \sqsubseteq \texttt{s}$. This ordering reflects our intuition about the termination behavior of these transformations: unfolding a function call will terminate less often than its specialization. This means that replacing a call unfolding by suspension cannot cause non-termination; however, the converse is not true. A detailed discussion on the treatment of calls can be found in [Sestoft, 1988], for example.

The way a function is specialized is specified by the second part of the filter. It has the functionality ($\widetilde{\textbf{Values}}^n \rightarrow \widetilde{\textbf{Values}}^n$). It receives the binding-time value of each argument of the call and returns a list of binding-time values; each of which specifies if the corresponding argument should be propagated. Specifically, value *static* indicates that the corresponding argument is to be propagated, whereas value *dynamic* indicates otherwise. As an example, function `appendn` with filter can be written as follows:

```
fun appendn(n,m,l) =
    (filter (if (stat? n) then u else s) [n,Dyn,l])
    if (n < 0) then l else m :: appendn(n-1,m+1,l)
```

where the two components of the specified filter are the bodies of the two filter functions described above. During facet computation, each parameter (such as `n`, `m` and `l`) is assigned the binding-time value of its corresponding argument value.

In the first component of the filter, predicate `stat?` returns true if its argument is static. The above filter specifies that whenever the first argument of a call to function **appendn** is static, the function call should be unfolded. Otherwise, it should be suspended. In specializing the function, the second component of the filter specifies that the second parameter is assigned a binding-time value *dynamic* (`Dyn`), and the corresponding argument is therefore not propagated.

keeping filter specification local to each function is crucial in controlling the partial-evaluation process with respect to the context of a call. Consider the filter of `appendn` above. It expresses different behaviors: if `appendn` is called with the first argument (parameter `n`) static then the call is unfold; otherwise, the call is suspend, and `appendn` is specialized without propagating the second argument. This contrasts with some of the existing strategies which provide unconditional annotation at each call (or function) to indicate how that call is (or all calls to that function are) to be treated [Haraldsson, 1977,Jones *et al.*, 1985]. In these systems, an annotation denotes an unconditional directive to the partial evaluator. Further discussion can be found in [Consel, 1988,Consel, 1989].

Note that one could introduce an automatic phase to annotate a program as to what to do for each function call. However, these annotations may lower the quality of the residual programs and can sometimes cause non-termination [Sestoft, 1988].

Although the above discussion applies to conventional partial evaluation, it can be readily applied to the PPE.[6] Thus, for a function f, the two components of its filter are denoted by $Ft[\![f]\!]\downarrow 1$ and $Ft[\![f]\!]\downarrow 2$ respectively. When a function call is suspended, a specialized function will be created. The specialized-function name is denoted by f_i^{sp}. It is uniquely identified by two components: the name of the original function f_i and the *specialization pattern.*[7]

[6] Using other kind of facet information to control the treatment of function call can be achieved with minor extension to the filter. This is discussed in Section 3.6.

[7] The specialization pattern describes information about the arguments used in specializing the function. Each argument value is represented in the pattern by an expression and a product of facet values. The expression is either a constant (which is to be propagated at function specialization) or a parameter name (representing an unknown argument). Thus, a specialized pattern is defined as:

In the on-line semantic specification depicted above, we use function $\widehat{bt}$ to determine the binding-time value of a value in $Result_{\widehat{\mathcal{E}}}$. Formally, $\widehat{bt}$ is defined as $\widehat{bt}\ (e, \hat{\delta})$ $= \tilde{\tau}\ (\hat{\delta}^1)$.

Creation of specialization patterns and specialized-function names are achieved by continuous functions *SpPat* and *SpName* respectively. Function *SpPat* is defined as follows:

$$
\begin{array}{lcl}
SpPat & : & \mathbf{Fn} \times Result^n_{\widehat{\mathcal{E}}} \times \widetilde{\mathbf{Values}}^n \rightarrow Result^n_{\widehat{\mathcal{E}}} \quad and \\
SpPat & = & \lambda(f, \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \langle b_1, \ldots, b_n \rangle)\ .\ \langle \hat{v}'_1, \ldots, \hat{v}'_n \rangle \\
 & & where\ \forall\ i \in \{1, \ldots, n\}, \\
 & & \qquad \hat{v}'_i\ =\ \langle e'_i, \langle \hat{d}_i, \hat{\delta}^2_i, \ldots, \hat{\delta}^m_i \rangle \rangle \\
 & & \qquad \langle e'_i, \hat{d}_i \rangle\ =\ b_i = static\ \rightarrow\ \langle e_i, \hat{\delta}^1_i \rangle, \\
 & & \qquad\qquad\qquad b_i = dynamic\ \rightarrow\ \langle x_i, \top_{\widehat{Values}} \rangle,\ \langle \bot_{Exp}, \bot_{\widehat{Values}} \rangle \\
 & & \qquad \hat{v}_i\ =\ \langle e_i, \langle \hat{\delta}^1_i, \hat{\delta}^2_i, \ldots, \hat{\delta}^m_i \rangle \rangle
\end{array}
$$

where $x_1, \ldots, x_n$ are the parameters of function f. Functionally, *SpPat* converts some of the constant values in the call arguments into non-constant values (since not all constant values are propagated during function specialization). By converting a constant into a non-constant, *SpPat* replaces the expression component of that constant argument by a parameter, and raises the product-of-facet-value component to a coarser value (one whose partial-evaluation-facet value is $\top_{\widehat{Values}}$). Using the terminology of abstract interpretation, we say that the non-constant value is a *safe approximation* of the original constant value. This implies that *SpPat* satisfies the following property:

Property 3.1 *Let* $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ *be the arguments of a call to function* f. *For all* $i \in \{1, \ldots, n\}$, *let* $b_i \in \widetilde{\mathbf{Values}}$ *such that* $\widehat{bt}(\hat{v}_i) \sqsubseteq_{\widehat{Values}} b_i$. *Then*

$$\forall i \in \{1, \ldots, n\},\ \hat{v}_i{\downarrow}2\ \sqsubseteq_{\widehat{\mathcal{SD}}}\ \hat{v}'_i{\downarrow}2,$$

where $\langle \hat{v}'_1, \ldots, \hat{v}'_n \rangle = SpPat\ (f, \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \langle b_1, \ldots, b_n \rangle)$.

$(\mathbf{Exp} \times \widehat{\mathcal{SD}})^n$, or simply, $Result^n_{\widehat{\mathcal{E}}}$.

This property can be easily verified by noticing that only the partial-evaluation-facet value of the second component of any argument may change in the computation. Furthermore, such a change only makes the value coarser.

Function *SpName* produces a unique specialized-function name from the original function name and the specialization pattern. It has the functionality:

$$SpName \ : \ (\mathbf{Fn} \times Result^n_{\widehat{\mathcal{E}}}) \rightarrow \mathbf{SpFn}$$

where **SpFn** is a flat domain of specialized-function names.

Those call arguments that are *not* propagated during function specialization become arguments of the residual call. They are extracted from the call arguments by the continuous function *ResidArgs*. *ResidArgs* has the following functionality:

$$ResidArgs \ : \ \mathbf{Fn} \times \widetilde{\mathbf{Values}}^n \times \mathbf{Exp}^n \rightarrow \mathbf{Exp}^m \quad (for\ m \leq n).$$

Furthermore, *ResidPars* is a continuous function that returns a tuple of parameters; each parameter replaces a residual argument in forming the specialization pattern. *ResidPars* has the following functionality:

$$ResidPars \ : \ \mathbf{Fn} \times \mathbf{Exp}^n \rightarrow \mathbf{Var}^m \quad (for\ m \leq n).$$

3.4.3 The Global Semantic Function $\widehat{\mathcal{A}}$

Function $\widehat{\mathcal{A}}$ collects *partial-evaluation signatures* associated with user-defined functions. A partial-evaluation signature is created when a non-trivial function call is partially evaluated. It consists of two components: A transformation tag (from domain **Transf** defined in page 65) indicating the transformation performed on the function, and the argument values of the application. If the call is suspended, its corresponding partial-evaluation signature is a transformation tag together with a specialization pattern.

All signatures are recorded in a *cache*. Formally, it is defined as

$$\mathbf{Cache}_{\widehat{\mathcal{A}}} \ = \ \mathbf{Fn} \ \rightarrow \ \mathcal{P}(\mathbf{Transf} \times Result^n_{\widehat{\mathcal{E}}}).$$

The cache is updated using a lub operation which is equivalent to set-union operation. That is, $\forall \sigma_1, \sigma_2 \in \mathbf{Cache}_{\widehat{\mathcal{A}}}$, $\sigma_1 \sqcup \sigma_2 = \lambda f \,.\, (\sigma_1[\![f]\!] \cup \sigma_2[\![f]\!])$.

Lastly, it is worth noticing that, just like a binding-time analysis, $\widehat{\mathcal{E}}_{Prog}$ performs a fixed-point iteration to obtain a cache. Such fixed-point iteration can be viewed as a semantic specification of the pending-list technique used in existing partial evaluators. The cache produced will be used by *MkProg* to generate the residual code for all the specialized functions.

The following lemma state the continuity property of both $\widehat{\mathcal{E}}$ and $\widehat{\mathcal{A}}$. The proof for the continuity of $\widehat{\mathcal{E}}$ is done by induction, whereas the proof for that of $\widehat{\mathcal{A}}$ is achieved again using the technique described in [Hudak and Young, 1991]. We omit the detail here.

Lemma 3.2 *Both $\widehat{\mathcal{E}}$ and $\widehat{\mathcal{A}}$ are continuous in all their arguments.*

3.4.4 Correctness of the PPE Semantics

Before proving the correctness of the semantics, we can already show that PPE semantics subsumes standard evaluation in the following sense:

Theorem 3.2 *Given a program P, suppose that (1) the input to this program is completely known at partial-evaluation time, and (2) all function calls in this program are unfolded during partial evaluation, then for any expression e in P,*

$$\hat{\tau}(\mathcal{E}\ [\![e]\!](\rho, \phi)) = (\widehat{\mathcal{E}}\ [\![e]\!](\hat{\rho}, \hat{\phi}))\downarrow 1,$$

where both $\phi \in FunEnv$ and $\hat{\phi} \in \widehat{FunEnv}$ are fixed for the program, $\rho \in VarEnv$, and $\hat{\rho} \in \widehat{VarEnv}$ is defined as:

$$\hat{\rho} = \lambda\, [\![x]\!] \,.\, \langle \hat{\tau}(d), \langle \hat{\alpha}_{\widehat{\mathcal{D}}^1}(d), \ldots, \hat{\alpha}_{\widehat{\mathcal{D}}^m}(d) \rangle \rangle \quad \textit{where } d = (\rho[\![x]\!]) \in \mathbf{D}.$$

Proof : We need to show that $\mathcal{E}[\![e]\!]\ \widehat{\mathcal{R}}\ \hat{\mathcal{E}}[\![e]\!]$, for the logical relation $\widehat{\mathcal{R}}$ between domains of $\mathcal{E}$ and $\hat{\mathcal{E}}$ defined by:

$$v\ \widehat{\mathcal{R}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{v} \quad \Leftrightarrow \quad \hat{\tau}(v) = \hat{v}{\downarrow}1 \wedge \hat{v}{\downarrow}2 = \langle \hat{\alpha}_{\widehat{\mathcal{D}}^1}(v), \ldots, \hat{\alpha}_{\widehat{\mathcal{D}}^m}(v) \rangle \ \ where\ v \in \mathbf{D}$$

$$\rho\ \widehat{\mathcal{R}}_{VarEnv}\ \hat{\rho} \quad \Leftrightarrow \quad \forall [\![x]\!] \in \mathbf{Var},\ \rho[\![x]\!]\ \widehat{\mathcal{R}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{\rho}[\![x]\!]$$

$$\langle d_1, d_2 \rangle\ \widehat{\mathcal{R}}_{D_1 \times D_2}\ \langle \hat{d}_1, \hat{d}_2 \rangle \quad \Leftrightarrow \quad d_1\ \widehat{\mathcal{R}}_{D_1}\ \hat{d}_1 \wedge d_2\ \widehat{\mathcal{R}}_{D_2}\ \hat{d}_2$$

$$f\ \widehat{\mathcal{R}}_{D_1 \to D_2}\ \hat{f} \quad \Leftrightarrow \quad \forall d \in D_1, \forall \hat{d} \in \widehat{D}_1,\ d\ \widehat{\mathcal{R}}_{D_1}\ \hat{d} \quad \Rightarrow \quad f(d)\ \widehat{\mathcal{R}}_{D_2}\ \hat{f}(\hat{d})$$

It suffices to show that $\phi\ \widehat{\mathcal{R}}\ \hat{\phi}$. Since this involves the recursive function environments ϕ and $\hat{\phi}$, we prove the validity of $\widehat{\mathcal{R}}$ using fixed-point induction on Kleene's chain over ϕ and $\hat{\phi}$, with the least element (in this proof, i ranges over all user-defined functions):

$$\begin{aligned}\langle \phi_0, \hat{\phi}_0 \rangle \ = \ \langle\ & \perp[(strict\ (\lambda(v_1, \ldots, v_n)\ .\ \perp_{Values})/f_i], \\ & \perp[(strict\ (\lambda(\hat{v}_1, \ldots, \hat{v}_n)\ .\ \langle \perp_{Exp}, \perp_{\widehat{\mathcal{SD}}} \rangle)/f_i] \rangle.\end{aligned}$$

It is true trivially that $\phi_0\ \widehat{\mathcal{R}}\ \hat{\phi}_0$.

Suppose that $\widehat{\mathcal{R}}$ is true for some element $\langle \phi_n, \hat{\phi}_n \rangle$ in the ascending chain, we would like to prove that $\widehat{\mathcal{R}}$ is true for $\langle \phi_{n+1}, \hat{\phi}_{n+1} \rangle$ where

$$\begin{aligned}\langle \phi_{n+1}, \hat{\phi}_{n+1} \rangle \ = \ \langle\ & \perp[(strict\ \{\lambda(v_1, \ldots, v_n)\ .\ \mathcal{E}[\![e_i]\!](\perp[v_k/x_k], \phi_n)\}/f_i], \\ & \perp[(strict\ \{\lambda(\hat{v}_1, \ldots, \hat{v}_n)\ .\ \hat{\mathcal{E}}[\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}_n)\}/f_i] \rangle.\end{aligned}$$

That is, we want to show that

$\forall [\![f_i]\!] \in \mathbf{Fn}, \forall j \in \{1, \ldots, n\}, \forall v_j \in \mathbf{Values}, \forall \hat{v}_j \in Result_{\widehat{\mathcal{E}}}$,

$$\bigwedge_{j=1}^{n} (v_j\ \widehat{\mathcal{R}}\ \hat{v}_j) \quad \Rightarrow \quad \phi_{n+1}[\![f_i]\!](v_1, \ldots, v_n)\ \widehat{\mathcal{R}}\ \hat{\phi}_{n+1}[\![f_i]\!](\hat{v}_1, \ldots, \hat{v}_n).$$

Or, equivalently,

$\forall [\![f_i]\!] \in \mathbf{Fn}, \forall j \in \{1, \ldots, n\}, \forall v_j \in \mathbf{Values}, \forall \hat{v}_j \in Result_{\widehat{\mathcal{E}}}$,

$$\bigwedge_{j=1}^{n} (v_j\ \widehat{\mathcal{R}}\ \hat{v}_j) \quad \Rightarrow \quad \mathcal{E}[\![e_i]\!](\perp[v_k/x_k], \phi_n)\ \widehat{\mathcal{R}}\ \hat{\mathcal{E}}[\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}_n).$$

The proof is by structural induction on e. It suffices to show that $\widehat{\mathcal{R}}$ holds for all the corresponding pairs of combinators used by $\mathcal{E}$ and $\hat{\mathcal{E}}$ respectively.

- The proofs for $Const_{\mathcal{E}}$ and $VarLookup_{\mathcal{E}}$ are easy, and thus ignored.

- $PrimOp_{\mathcal{E}}$: This is done by structural induction and a case analysis over all the possible argument values of the primitive.

- $Cond_{\mathcal{E}}$: This is done by structural induction and a case analysis over the possible values produced by $\hat{k}_1(\perp[\hat{v}_k/x_k], \hat{\phi}_n)$.

- $App_{\mathcal{E}}$: For any user-defined function f, suppose that all corresponding arguments of $App_{\mathcal{E}}$ and $App_{\widehat{\mathcal{E}}}$ are related by $\widehat{\mathcal{R}}$. $\forall i \in \{1, \ldots, n\}$, let $v_i = k_i(\perp[v_k/x_k], \phi_n)$ and $\hat{v}_i = \hat{k}_i(\perp[\hat{v}_k/x_k], \hat{\phi}_n)$. We have $\forall i \in \{1, \ldots, n\}$, $v_i \, \widehat{\mathcal{R}}_{Result_{\widehat{\mathcal{E}}}} \, \hat{v}_i$ by the structural-induction hypothesis. Since both ϕ_n and $\hat{\phi}_n$ contain only strict functions, $\widehat{\mathcal{R}}$ holds when some of the arguments is bottom. On the other hand, under the condition that all function applications are unfolded, $App_{\widehat{\mathcal{E}}}[\![f]\!](\hat{k}_1, \ldots, \hat{k}_n)$ is reduced to $\hat{\phi}_n \; [\![f]\!] \; (\hat{v}_1, \ldots, \hat{v}_n)$. By the fixed-point-induction hypothesis,

$$\phi_n \, \widehat{\mathcal{R}} \, \hat{\phi}_n \quad \Rightarrow \quad \phi_n[\![f]\!](v_1, \ldots, v_n) \, \widehat{\mathcal{R}} \, \hat{\phi}_n[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n).$$

  Hence, $App_{\mathcal{E}} \, \widehat{\mathcal{R}} \, App_{\widehat{\mathcal{E}}}$.

This concludes the proof. □

We begin the discussion about the correctness of PPE by first observing that any constant produced by partially evaluating a primitive call is always correct with respect to the standard semantics, modulo termination. This observation is stated formally in the following lemma:

Lemma 3.3 *Let* $[\widehat{\mathcal{D}}; \widehat{\Omega}]$ *be a product of facets for an algebra* $[\mathbf{D}; \mathbf{O}]$. *Let* $c = (\widehat{\mathcal{E}}[\![p(x_1, \ldots, x_n)]\!] \; (\perp[\langle [\![x_i]\!], \hat{\delta}_i \rangle / x_i], \perp)) \downarrow 1$, *and* $v = \mathcal{E}[\![p(x_1, \ldots, x_n)]\!](\perp[d_i/x_i], \perp)$ *where* $d_i \in \bigcap_{j=1}^{m} \{d \in \mathbf{D} \mid d \preceq_{\widehat{\alpha}_{\widehat{\mathcal{D}}_j}} \hat{\delta}_i^j\}$, $\forall i \in \{1, \ldots, n\}$. *Then,*

$$(c \in \mathbf{Const}) \; and \; (v \neq \perp_{Values}) \quad \Rightarrow \quad c = \widehat{\tau}(v).$$

Proof : First, we notice that

$$(\mathcal{E}[\![p(x_1,\ldots,x_n)]\!]\ (\bot[d_i/x_i],\bot)) \;=\; \mathcal{K}_P[\![p]\!](d_1,\ldots,d_n) \;=\; p(d_1,\ldots,d_n).$$

As defined in the on-line PPE semantics, we have

$$(\widehat{\mathcal{E}}[\![p(x_1,\ldots,x_n)]\!](\bot[\langle[\![x_i]\!],\hat{\delta}_i\rangle/x_i],\bot))\downarrow 1 \;=(\widehat{\mathcal{K}}_P[\![p]\!](\langle[\![x_1]\!],\hat{\delta}_1\rangle,\ldots,\langle[\![x_n]\!],\hat{\delta}_n\rangle))\downarrow 1.$$

Given that $(\widehat{\mathcal{E}}[\![p(x_1,\ldots,x_n)]\!](\bot[\langle[\![x_i]\!],\hat{\delta}_i\rangle/x_i],\bot))\downarrow 1 \in \mathbf{Const}$, the proof is done by case analysis of the different classes of primitive operators:

1. If p is a closed operator:

   Given $(\widehat{\mathcal{K}}_P[\![p]\!](\langle[\![x_1]\!],\hat{\delta}_1\rangle,\ldots,\langle[\![x_n]\!],\hat{\delta}_n\rangle)\downarrow 1 \in \mathbf{Const}$. By definition of on-line PPE for closed operators, this constant can only be produced by the partial-evaluation facet (that is, the first component of the product of facets). Thus,

   $$(\widehat{\mathcal{K}}_P[\![p]\!](\langle[\![x_1]\!],\hat{\delta}_1\rangle,\ldots,\langle[\![x_n]\!],\hat{\delta}_n\rangle)\downarrow 1 = \hat{p}_1(\hat{\delta}_1^1,\ldots,\hat{\delta}_n^1) \in \mathbf{Const}.$$

   Given that $(\mathcal{E}[\![p(x_1,\ldots,x_n)]\!](\bot[d_i/x_i],\bot) \neq \bot$ and $\hat{p}_1(\hat{\delta}_1^1,\ldots,\hat{\delta}_n^1) \in \mathbf{Const}$, by Definition 2.9, we have $\forall i \in \{1,\ldots,n\}, \hat{\delta}_i^1 \in \mathbf{Const}$. Then $\forall i \in \{1,\ldots,n\}, \forall d_i \in \mathbf{D}$ such that $d_i = \mathcal{K}(\hat{\delta}_i^1)$:

   $$\hat{p}_1(\hat{\delta}_1^1,\ldots,\hat{\delta}_n^1) \;=\; \hat{\tau}(\mathcal{K}_P[\![p]\!](d_1,\ldots,d_n)) \;=\; \hat{\tau}(p(d_1,\ldots,d_n)).$$

2. If p is an open operator:

   $(\widehat{\mathcal{K}}_P[\![p]\!](\langle[\![x_1]\!],\hat{\delta}_1\rangle,\ldots,\langle[\![x_n]\!],\hat{\delta}_n\rangle))\downarrow 1 \in \mathbf{Const}$ implies that this constant is produced by a facet operation in the product of facets. Lemma 2.1 says that we can consider any facet that produces the constant. Assume that the i^{th} facet produces this constant; This is denoted by $\hat{p}_i(\hat{\delta}_1^i,\ldots,\hat{\delta}_n^i)$. By Property 2.2, we have $\hat{p}_i(\hat{\delta}_1^i,\ldots,\hat{\delta}_n^i) \;=\; \hat{\tau}(p(d_1,\ldots,d_n))$.

This concludes the proof. □

Since an abstract value used during partial evaluation represents a set of concrete values, a partially-known input $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ to a program during partial evaluation represents a set of concrete inputs to that program. That is,

$$\langle \hat{v}_1, \ldots, \hat{v}_n \rangle \textit{ represents the set } \{\langle v_1, \ldots, v_n \rangle \mid \hat{\alpha}_{\widehat{\mathcal{D}}_i}(v_i) = \hat{v}_i,\ i \in \{1, \ldots, n\}\},$$

where, for each $v_i \in \mathbf{D}_i$, the corresponding abstraction function is $\hat{\alpha}_{\widehat{\mathcal{D}}_i}$ (for $i \in \{1, \ldots, n\}$). The safety criterion described in Equation 3.1 is now expressed in our semantic specification as follows: Partial evaluation of a program with input $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ is correct if it produces a cache that captures all possible non-trivial calls performed during the execution of a program (under the instrumented semantics) with input taken from the set represented by $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$, modulo termination.

One word about termination. All nontrivial (conventional) partial evaluators so far have had problems in preserving termination. The problem can either be with the partial-evaluation process itself or with the generated residual program. (See [Gomard, 1992] for discussion about termination preservation.) This problem also exists with PPE. In fact, since the termination of call unfolding/specialization is controlled by user-annotation (via filter specification), we rely on the user to ensure the termination of the partial-evaluation process. The safety criterion expressed above is the same as that expressed for conventional partial evaluation of applicative programs.

The safety of the semantics can be shown by relating the local and global semantics to their respective counterpart in the instrumented semantics. That is, we define a logical relation $\mathcal{R}^{\widehat{\mathcal{E}}}$ relating $\mathcal{E}$ and $\widehat{\mathcal{E}}$, and a logical relation $\mathcal{R}^{\widehat{\mathcal{A}}}$ relating $\mathcal{A}$ and $\widehat{\mathcal{A}}$. Notice that $\mathcal{R}^{\widehat{\mathcal{E}}}$ relates the results v and $\hat{v}$ computed by $\mathcal{E}$ and $\widehat{\mathcal{E}}$ respectively. Since $\hat{v} = \langle e, \hat{\delta} \rangle \in (\mathbf{Exp} \times \widehat{\mathcal{SD}})$, $\mathcal{R}^{\widehat{\mathcal{E}}}$ is composed of two relations, $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ and $\mathcal{R}^{\widehat{\mathcal{E}}_2}$, that relate a concrete value v to e and $\hat{\delta}$ respectively. It turns out that the correctness of $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ depends on that of $\mathcal{R}^{\widehat{\mathcal{A}}}$. At the same time, the correctness of $\mathcal{R}^{\widehat{\mathcal{A}}}$ depends partly on the correctness of $\mathcal{R}^{\widehat{\mathcal{E}}_2}$. Therefore, we shall prove the correctness of $\mathcal{R}^{\widehat{\mathcal{E}}_2}$, then that of $\mathcal{R}^{\widehat{\mathcal{A}}}$, and finally that of $\mathcal{R}^{\widehat{\mathcal{E}}_1}$. Lastly, we combine the result of $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ and $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ to express the correctness of $\mathcal{R}^{\widehat{\mathcal{E}}}$.

Correctness of $\mathcal{R}^{\widehat{\mathcal{E}}_2}$

In this section, we define and prove the correctness of the relation $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ between the result of $\mathcal{E}$ and the second component (the product of facets) of the result of $\widehat{\mathcal{E}}$. We begin with a relation which relates a basic value with a product-of-facet value:

Definition 3.1 (Relation $\preceq_{\hat{\alpha}_{\widehat{SD}}}$) *For any value $v \in \mathbf{D}$ and $\hat{\delta} \in \widehat{SD}$ with $\hat{\delta} = \langle \hat{\delta}^1, \ldots, \hat{\delta}^m \rangle$,*

$$v \preceq_{\hat{\alpha}_{\widehat{SD}}} \hat{\delta} \;\Leftrightarrow\; \forall i \in \{1, \ldots, m\},\; v \preceq_{\hat{\alpha}_{\widehat{D}^i}} \hat{\delta}^i,$$

where $\preceq_{\hat{\alpha}_{\widehat{D}^i}}$ is the logical relation induced from the facet mapping from $\mathbf{D}$ to $\widehat{\mathbf{D}}^i$, and $\widehat{\mathbf{D}}^1 \otimes \cdots \otimes \widehat{\mathbf{D}}^m$ is the product of facets of $\mathbf{D}$.

Since $(v \preceq_{\hat{\alpha}_{\widehat{SD}}} \hat{\delta}) \equiv (\bigwedge_{i=1}^{m} (v \preceq_{\hat{\alpha}_{\widehat{D}^i}} \hat{\delta}^i))$, $\preceq_{\hat{\alpha}_{\widehat{SD}}}$ is a logical relation between **Values** and $\widehat{SD}$ (assuming that the values have been injected into their respective sum domains).

Next, relation $\preceq_{\hat{\alpha}_{\widehat{SD}}}$ is extended to provide relationship between various domains used by the local semantic functions of both the instrumented and the on-line PPE semantics.

Definition 3.2 (Relation $\mathcal{R}^{\widehat{\mathcal{E}}_2}$) *$\mathcal{R}^{\widehat{\mathcal{E}}_2}$ is a logical relation between domains of $\mathcal{E}$ and $\widehat{\mathcal{E}}$ defined by:*

$$\begin{aligned}
v \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{Result_{\widehat{\mathcal{E}}}}\; \hat{v} \;&\Leftrightarrow\; v \sqsubseteq_{\hat{\alpha}_{\widehat{SD}}} \hat{v}\downarrow 2,\\
\rho \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{VarEnv}\; \hat{\rho} \;&\Leftrightarrow\; \forall [\![x]\!] \in \mathbf{Var},\; \rho[\![x]\!] \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{Result_{\widehat{\mathcal{E}}}}\; \hat{\rho}[\![x]\!],\\
\phi \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{FunEnv}\; \hat{\phi} \;&\Leftrightarrow\; \forall [\![f]\!] \in \mathbf{Fn}, \forall i \in \{1, \ldots, n\}, \forall v_i \in \mathbf{Values}, \forall \hat{v}_i \in Result_{\widehat{\mathcal{E}}},\\
&\qquad \bigwedge_{i=1}^{n} (v_i \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{Result_{\widehat{\mathcal{E}}}}\; \hat{v}_i) \;\Rightarrow\; \phi[\![f]\!](v_1, \ldots, v_n) \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{Result_{\widehat{\mathcal{E}}}} \hat{\phi}[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n)\\
\langle d_1, d_2 \rangle \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_1 \times D_2}\; \langle \hat{d}_1, \hat{d}_2 \rangle \;&\Leftrightarrow\; d_1 \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_1}\; \hat{d}_1 \;\wedge\; d_2 \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_2}\; \hat{d}_2\\
f \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_1 \to D_2}\; \hat{f} \;&\Leftrightarrow\; \forall d \in D_1, \forall \hat{d} \in \widehat{D}_1,\; d \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_1}\; \hat{d} \;\Rightarrow\; f(d) \;\mathcal{R}^{\widehat{\mathcal{E}}_2}_{D_2}\; \hat{f}(\hat{d}).
\end{aligned}$$

The next lemma shows that the use of user-defined functions at on-line level is related to that at the instrumented semantics.

Lemma 3.4 *Given a program P, let ϕ and $\hat{\phi}$ be the two function environments for P defined by the standard and the PPE semantics respectively. Then $\phi \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ \hat{\phi}$.*

Proof : We need to show that $\forall [\![f]\!] \in \mathbf{Fn}, \forall i \in \{1, \ldots, n\}, \forall v_i \in \mathbf{Values}, \forall \hat{v}_i \in Result_{\widehat{\mathcal{E}}}$,

$$\bigwedge_{i=1}^{n} (v_i \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ \hat{v}_i) \ \Rightarrow \ \phi[\![f]\!](v_1, \ldots, v_n) \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ \hat{\phi}[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n).$$

The proof is similar to that for Theorem 3.2, proving the validity of relation $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ using fixed-point induction on Kleene's chain over ϕ and $\hat{\phi}$. $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ trivially holds for the least element in the chain.

Suppose that $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ is true for some element $\langle \phi_n, \hat{\phi}_n \rangle$ in the ascending chain, we would like to prove that $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ is true for $\langle \phi_{n+1}, \hat{\phi}_{n+1} \rangle$ where

$$\begin{aligned} \langle \phi_{n+1}, \hat{\phi}_{n+1} \rangle \ = \ \langle \ & \perp[(strict \ \{\lambda(v_1, \ldots, v_n) \ . \ \mathcal{E}[\![e_i]\!](\perp[v_k/x_k], \phi_n)\}/f_i], \\ & \perp[(strict \ \{\lambda(\hat{v}_1, \ldots, \hat{v}_n) \ . \ \widehat{\mathcal{E}}[\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}_n)\}/f_i]\rangle. \end{aligned}$$

That is, we want to show that
$\forall [\![f]\!] \in \mathbf{Fn}, \forall j \in \{1, \ldots, n\}, \forall v_j \in \mathbf{Values}, \forall \hat{v}_j \in Result_{\widehat{\mathcal{E}}}$,

$$\bigwedge_{j=1}^{n} (v_j \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ \hat{v}_j) \ \Rightarrow \ \phi_{n+1}[\![f]\!](v_1, \ldots, v_n) \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ \hat{\phi}_{n+1}[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n).$$

The proof is by structural induction on e. It suffices to show that $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ holds for all the corresponding pairs of combinators used by $\mathcal{E}$ and $\widehat{\mathcal{E}}$ respectively.

- $Const_{\mathcal{E}}$: $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ is true trivially by comparing $\mathcal{K}$ and $\widehat{\mathcal{K}}$.
- $VarLookup_{\mathcal{E}}$: by structural induction.
- $PrimOp_{\mathcal{E}}$: $PrimOp_{\mathcal{E}} \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ PrimOp_{\widehat{\mathcal{E}}}$ holds by structural induction and a case analysis over the values produced by $PrimOp_{\widehat{\mathcal{E}}}$. Proof is omitted.
- $Cond_{\mathcal{E}}$: $Cond_{\mathcal{E}} \ \mathcal{R}^{\widehat{\mathcal{E}}_2} \ Cond_{\widehat{\mathcal{E}}}$ holds by structural induction and a case analysis over the values produced by $\hat{k}_1(\hat{\rho}, \hat{\phi}_n)$.

- $App_{\mathcal{E}}$: For any user-defined function f, all the corresponding arguments of $App_{\mathcal{E}}$ and $App_{\widehat{\mathcal{E}}}$ are related by $\mathcal{R}^{\widehat{\mathcal{E}}_2}$ (by the structural-induction hypothesis).

  If the call is unfolded, $App_{\widehat{\mathcal{E}}}[\![f]\!]\ (\hat{k}_1, \ldots,\ \hat{k}_n)$ is reduced to $\hat{\phi}_n[\![f]\!]\ (\hat{v}_1, \ldots, \hat{v}_n)$, whereas $App_{\mathcal{E}}[\![f]\!]\ (k_1, \ldots,\ k_n)$ is reduced to $\phi_n[\![f]\!]\ (v_1, \ldots, v_n)$. Notice that $v_i \mathcal{R}^{\widehat{\mathcal{E}}_2} \hat{v}_i\ \forall i \in \{1, \ldots, n\}$. Since $\phi_n\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \hat{\phi}_n$ by fixed-point-induction hypothesis, we have

  $$\phi_n[\![f]\!](v_1, \ldots, v_n)\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \hat{\phi}_n[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n).$$

  If the call is suspended, the second component of the result is $\hat{\phi}_n[\![f]\!]\ (\hat{v}'_1, \ldots, \hat{v}'_n)$. By Property 3.1 of function *SpPat*, we must have $\forall i \in \{1, \ldots, n\}, \hat{v}_i{\downarrow}2 \sqsubseteq_{\widehat{SD}} \hat{v}'_i{\downarrow}2$ and therefore $v_i\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \hat{v}'_i$. Let $\hat{v} = \hat{\phi}_n[\![f]\!](\hat{v}_1, \ldots, \hat{v}_n)$. We have

  $$\phi_n[\![f]\!](v_1, \ldots, v_n) \preceq_{\hat{\alpha}_{\widehat{SD}}}\ \hat{v} \quad \text{[fixed-point-induction hypothesis]}$$

  and

  $$\hat{v}{\downarrow}2 \sqsubseteq_{\widehat{SD}}\ (\hat{\phi}_n[\![f]\!](\hat{v}'_1, \ldots, \hat{v}'_n)){\downarrow}2 \quad \text{[Continuity of } \hat{\phi}_n[\![f]\!]\text{]}$$

  Thus, $\phi_n[\![f]\!](v_1, \ldots, v_n)\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \hat{\phi}_n[\![f]\!](\hat{v}'_1, \ldots, \hat{v}'_n)$. Hence, $App_{\mathcal{E}}\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ App_{\widehat{\mathcal{E}}}$.

Hence, $\phi\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \hat{\phi}$. This concludes the proof. □

Lemma 3.4 leads directly to Theorem 3.3, which states that the local semantic function $\widehat{\mathcal{E}}$ is an "abstraction" of the $\mathcal{E}$ in the sense that, if the abstract values accepted by $\widehat{\mathcal{E}}$ is a safe approximation of the static properties derived from the basic values accepted by $\mathcal{E}$, then the result produced by $\widehat{\mathcal{E}}$ is also a safe approximation of the static properties produced by $\mathcal{E}$.

Theorem 3.3 (Correctness of Local Semantics – 2nd Component) $\mathcal{E}\ \mathcal{R}^{\widehat{\mathcal{E}}_2}\ \widehat{\mathcal{E}}$.

Proof : From Lemma 3.4. □

Before we close this section, let us make an observation about the relationship between the first and second components of a value produced by $\widehat{\mathcal{E}}$.

Observation 3.1 *During partial evaluation, all values $\hat{v} \in Result_{\widehat{\mathcal{E}}}$ satisfy the following conditions:* 1. $\hat{v}{\downarrow}1 \in \mathbf{Const} \wedge (\hat{v}{\downarrow}2){\downarrow}1 \in \mathbf{Const} \Leftrightarrow \hat{v}{\downarrow}1 = (\hat{v}{\downarrow}2){\downarrow}1$;

2. $\hat{v}{\downarrow}1 = \bot_{Exp} \Leftrightarrow (\hat{v}{\downarrow}2){\downarrow}1 = \bot_{\widehat{Values}}$.

The above observation comes directly from the definitions of $\widehat{\mathcal{K}}$ and $\widehat{\mathcal{K}}_P$ in Figure 3.9.

We say that a value $\hat{v} \in Result_{\widehat{\mathcal{E}}}$ is $\mathcal{R}$-*consistent* if (1) it satisfies one of the above conditions and (2) its second component (product of facets) is *consistent*, as defined in Definition 2.8. This fact is used in the next section.

Correctness of the Global Semantics

In this section, we prove the correctness of the global partial-evaluation semantics (1) by relating the semantics of $\widehat{\mathcal{A}}$ with $\mathcal{A}$ using logical relation $\mathcal{R}^{\widehat{\mathcal{A}}}$, and (2) by showing that all the non-trivial calls performed at standard evaluation are captured by $\widehat{\mathcal{A}}$.

Since both $\mathcal{A}$ and $\widehat{\mathcal{A}}$ produce caches, $\mathcal{R}^{\widehat{\mathcal{A}}}$ relate these caches. That is, whenever a standard signature for a function is recorded in the cache produced by $\mathcal{A}$, there exists a logically related partial-evaluation signature for that function in the cache produced by $\widehat{\mathcal{A}}$. Formally,

Definition 3.3 (Relation $\mathcal{R}^{\widehat{\mathcal{A}}}$) $\mathcal{R}^{\widehat{\mathcal{A}}}$ *is a logical relation between domains of $\mathcal{A}$ and $\widehat{\mathcal{A}}$ defined by:*

$$v\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{v} \quad \Leftrightarrow \quad (\hat{v}\ is\ \mathcal{R}{-}consistent)\ \wedge\ (v\ \mathcal{R}^{\widehat{\mathcal{E}_2}}\ \hat{v})$$

$$\langle v_1,\ldots,v_n\rangle\ \mathcal{R}^{\widehat{\mathcal{A}}}_{(Transf\ \times\ Result^n_{\widehat{\mathcal{E}}})}\ \langle t,\hat{v}_1,\ldots,\hat{v}_n\rangle \quad \Leftrightarrow \quad \bigwedge_{i=1}^{n}(v_i\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{v}_i)$$

$$\sigma\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{A}}}}\ \hat{\sigma} \quad \Leftrightarrow \quad \forall[\![f]\!] \in Dom(\sigma), \forall s \in \sigma[\![f]\!], \exists \hat{s} \in \hat{\sigma}[\![f]\!],\ s\ \mathcal{R}^{\widehat{\mathcal{A}}}_{(Transf\ \times\ Result^n_{\widehat{\mathcal{E}}})}\ \hat{s}$$

$$\rho\ \mathcal{R}^{\widehat{\mathcal{A}}}_{VarEnv}\ \hat{\rho} \quad \Leftrightarrow \quad \forall[\![x]\!] \in \mathbf{Var},\ \rho[\![x]\!]\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{\rho}[\![x]\!]$$

$$\phi\ \mathcal{R}^{\widehat{\mathcal{A}}}_{FunEnv}\ \hat{\phi} \quad \Leftrightarrow \quad \forall[\![f]\!] \in \mathbf{Fn}, \forall j \in \{1,\ldots,n\}, \forall v_j \in \mathbf{Values}, \forall \hat{v}_j \in Result_{\widehat{\mathcal{E}}},$$
$$\bigwedge_{j=1}^{n}(v_j\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{v}_j)\ \Rightarrow\ \phi[\![f]\!](v_1,\ldots,v_n)\ \mathcal{R}^{\widehat{\mathcal{A}}}_{Result_{\widehat{\mathcal{E}}}}\ \hat{\phi}[\![f]\!](\hat{v}_1,\ldots,\hat{v}_n)$$

$$\langle d_1,d_2\rangle\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_1\times D_2}\ \langle \hat{d}_1,\hat{d}_2\rangle \quad \Leftrightarrow \quad d_1\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_1}\ \hat{d}_1\ \wedge\ d_2\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_2}\ \hat{d}_2$$

$$f\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_1\to D_2}\ \hat{f} \quad \Leftrightarrow \quad \forall d \in D_1, \forall \hat{d} \in \widehat{D}_1,\ d\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_1}\ \hat{d}\ \Rightarrow\ f(d)\ \mathcal{R}^{\widehat{\mathcal{A}}}_{D_2}\ \hat{f}(\hat{d}).$$

Note that the $\mathcal{R}$-consistency (Observation 3.1) ensures that the first component of $\hat{v}$, the residual expression, is consistent with the result of the partial-evaluation facet. Observe that there is no value in the standard signature corresponding to the transformation tag of the partial-evaluation signature. In fact, the transformation tag of a standard signature could have been obtained by performing filter computations at the standard-semantic level. However, the transformation has no effect on standard evaluation. Furthermore, since filters are continuous, the transformation tag computed is guaranteed to be at least as precise as that computed at the on-line level. Thus, we can ignore this information without compromising the correctness of the global semantics. Lastly, we note that the lub operation (which is the set-union operation) on caches preserves $\mathcal{R}^{\hat{\mathcal{A}}}$.

The next lemma shows that all the standard signatures recorded in the final cache produced by $\mathcal{A}$ are "captured" in the corresponding cache produced by $\hat{\mathcal{A}}$ in the sense that they are related by $\mathcal{R}^{\hat{\mathcal{A}}}$.

Notice that whenever $\hat{\mathcal{A}}$ uses a value $\hat{v}$ in decision making (in combinators $Cond_{\hat{\mathcal{A}}}$ and $App_{\hat{\mathcal{A}}}$), only the partial-evaluation-facet value is used, as is manifested by the definition of functions *SpPat* and $\hat{bt}$, and the description of function *Ft*. Therefore, only the second component of $\hat{v}$ is needed to show the correctness of $\hat{\mathcal{A}}$. Although the first component of $\hat{v}$ (the expression) is modified by $\hat{\mathcal{A}}$ when dealing with combinator $App_{\hat{\mathcal{A}}}$, it should be noted that the modification is the same as the one done in $\hat{\mathcal{E}}$, and by Observation 3.1, the modified value is still $\mathcal{R}$-consistent.

Lemma 3.5 *Given a program P, for any $\hat{\mathcal{E}}$ such that $\mathcal{E}\ \mathcal{R}^{\hat{\mathcal{A}}}\ \hat{\mathcal{E}}$, let ϕ and $\hat{\phi}$ be two function environments for P defined by the standard and the partial-evaluation semantics respectively. For any expression e in P, for any variable environments ρ and $\hat{\rho}$ such that $\rho\ \mathcal{R}^{\hat{\mathcal{A}}}\ \hat{\rho}$,*

$$\mathcal{A}[\![e]\!](\rho,\phi)\quad \mathcal{R}^{\hat{\mathcal{A}}}\quad \hat{\mathcal{A}}[\![e]\!](\hat{\rho},\hat{\phi}).$$

Proof : The proof is by structural induction on e. Firstly, notice that

$$\mathcal{E}\ \mathcal{R}^{\hat{\mathcal{A}}}\ \hat{\mathcal{E}}\ \Rightarrow\ \phi\ \mathcal{R}^{\hat{\mathcal{A}}}\ \hat{\phi}.$$

It suffices to show that $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds for all the corresponding pairs of combinators used by $\mathcal{A}$ and $\widehat{\mathcal{A}}$ respectively. By structural induction, it is easy to see that $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds for constants, variables and primitive calls.

1. $Cond_{\mathcal{A}}$: By structural-induction hypothesis, $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds for the test expression.

   (a) If $\hat{\delta}_1^1 \in \mathbf{Const}$, since $k(\rho, \phi)\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \hat{k}(\hat{\rho}, \hat{\phi})$, the branch chosen will be the same for both $\mathcal{A}$ and $\widehat{\mathcal{A}}$, and by structural-induction hypothesis, $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds.

   (b) If $\hat{\delta}_1^1 = \top_{\widehat{Values}}$, then all non-trivial calls in both branches are recorded by $\widehat{\mathcal{A}}$. Again, by structural-induction hypothesis, $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds.

2. $App_{\mathcal{A}}$: By structural-induction hypothesis, $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds for all the arguments to the application. As for the application itself, if it is recorded by $\mathcal{A}$, then it is non-trivial. By structural induction on the arguments, the application is also recorded by $\widehat{\mathcal{A}}$. Its transformation tag is either u or s. It is easy to see that $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds when the transformation tag is u. If the tag is s, $\mathcal{R}^{\widehat{\mathcal{A}}}$ holds if $\forall i \in \{1, \ldots, n\}$, $\hat{v}_i{\downarrow}2 \sqsubseteq \hat{v}'_i{\downarrow}2$, where $\langle \hat{v}'_1, \ldots, \hat{v}'_n \rangle$ is the result of applying *SpPat* in $\widehat{\mathcal{A}}$. This is true by Property 3.1.

Therefore, $\mathcal{A}[\![e]\!](\rho, \phi)\ \ \mathcal{R}^{\widehat{\mathcal{A}}}\ \ \widehat{\mathcal{A}}[\![e]\!](\hat{\rho}, \hat{\phi})$. □

Notice that, for a value $\hat{v}$, there may be more than one value v such that $v\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \hat{v}$. Therefore, the above lemma shows that given an expression e, $\widehat{\mathcal{A}}$ captures all calls within e that may be invoked under different initial value v with $v\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \hat{v}$. The following theorem uses Lemma 3.4 to prove that the final cache produced by the global semantics is "complete" in the sense that it captures all the non-trivial calls performed during standard evaluation.

Theorem 3.4 (Correctness of Global Partial-Evaluation Semantics) *Given a program P. Let $\widehat{\mathcal{E}}$ be a valuation function of the partial-evaluation semantics such that $\mathcal{E}\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \widehat{\mathcal{E}}$. Let $\langle v_1, \ldots, v_n \rangle$ and $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ be initial inputs to P for standard and partial-evaluation semantics respectively, such that $v_i\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \hat{v}_i$, $\forall i \in \{1, \ldots, n\}$. If σ and $\hat{\sigma}$ are the final caches produced by $\mathcal{A}$ and $\widehat{\mathcal{A}}$ respectively, then $\sigma\ \mathcal{R}^{\widehat{\mathcal{A}}}\ \hat{\sigma}$.*

Proof : Firstly, we notice from the definition of $\widehat{\mathcal{E}}_{Prog}$ that $\langle \mathbf{s}, \hat{v}_1, \ldots, \hat{v}_n \rangle \in \hat{\sigma}[\![f_1]\!]$, just like $\langle v_1, \ldots, v_n \rangle \in \sigma[\![f_1]\!]$. Next, $\hat{h}$ in $\widehat{\mathcal{E}}_{Prog}$ applies $\widehat{\mathcal{A}}$ to each partial-evaluation signature in the cache, similar to function h in $\mathcal{E}_{Prog}$. Since lub operation preserves $\mathcal{R}^{\widehat{\mathcal{A}}}$, $\sigma \; \mathcal{R}^{\widehat{\mathcal{A}}} \; \hat{\sigma}$. □

By Theorem 3.1, we know that σ contains all the non-trivial calls performed at the standard evaluation. Since $\sigma \; \mathcal{R}^{\widehat{\mathcal{A}}} \; \hat{\sigma}$, all these calls must be captured by $\hat{\sigma}$.

Correctness of $\mathcal{R}^{\widehat{\mathcal{E}}_1}$

We now prove the correctness of the residual expression produced by $\widehat{\mathcal{E}}$ using the relation $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ which relates a residual expression to a concrete value produced by $\mathcal{E}$. Intuitively, a residual expression and a concrete value are related if the former evaluates (under standard evaluation) to the latter. This requires "post-evaluation" of the residual expression. Therefore, $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ is not simply a relation between a value and a residual expression; it is a relation between a value, a residual expression and its "post-evaluated" value. In the following definition, we introduce the notion of *satisfiability* to aid in formulating this relation. This notion is similar, though simpler, to the definition of *agreeability* used by Gomard in [Gomard, 1992]. For clarity, $a =_\perp b$ denotes the equality between a and b, provided both of them terminate. Of Course, $a = b \Rightarrow a =_\perp b$. For an expression e, we define $FV(e)$ to be the set of free variables in e.

Definition 3.4 (Satisfiability) *Given a program P.Let $\rho_d \in VarEnv$ be a variable environment for a residual expression such that $Dom(\rho_d) = FV(\hat{v}{\downarrow}1)$. We say ρ_d* satisfies the pair $\langle v, \hat{v} \rangle$ *if*

$$v =_\perp \mathcal{E}[\![\hat{v}{\downarrow}1]\!](\rho_d, \phi') \; \wedge \; v \preceq_{\hat{\alpha}_{\widehat{SD}}} \hat{v}{\downarrow}2,$$

where $v \in$ **Values**, *$\hat{v} \in Result_{\widehat{\mathcal{E}}}$, and ϕ' is the function environment, defined by the standard semantics, for the specialized version of program P.*

Without loss of generality, we assume that every user-defined function consists of two parameters (x_1 and x_2). To show the relationship between $\mathcal{E}$ and $\widehat{\mathcal{E}}$, we must first show that the function environments they take as arguments are related. The following lemma clarifies this relation.

Lemma 3.6 *Given a program P. Let ϕ and $\hat{\phi}$ be the two function environments for P defined by the standard semantics and the partial-evaluation semantics respectively. For any user-defined function f, let ρ_d be a variable environment that satisfies both the pairs $\langle v_1, \hat{v}_1 \rangle$ and $\langle v_2, \hat{v}_2 \rangle$ with $v_1, v_2 \in$ **Values** and $\hat{v}_1, \hat{v}_2 \in Result_{\widehat{\mathcal{E}}}$. Then,*

$$\phi[\![f]\!]\ (v_1, v_2)\ =_\perp\ \mathcal{E}\ [\![(\hat{\phi}[\![f]\!]\ (\hat{v}_1, \hat{v}_2))\downarrow 1]\!]\ (\rho_d, \phi'),$$

where ϕ' is the function environment, defined by the standard semantics, for the specialized version of P.

Proof : The proof is similar to that described in [Gomard, 1992]. The major difference lies in the the fact that the property of a cache σ is used to show the correctness of partial evaluation of function application. This enables to show the correctness of multiple instances of specialized functions.

Since the lemma involves three functions *FunEnv*: ϕ, $\hat{\phi}$ for program P, and ϕ' for the residual program, we define the functional Φ as

$$\begin{aligned}\Phi\ \langle \phi_a, \hat{\phi}_a, \phi'_a \rangle\ =\ \langle\ & \perp[\{\lambda(v_1, v_2)\ .\ \mathcal{E}[\![e_i]\!](\perp[v_k/x_k,], \phi_a)\}/f_i \mid \forall\ [\![f_i]\!] \in \mathbf{Fn}], \\ & \perp[\{\lambda(\hat{v}_1, \hat{v}_2)\ .\ \widehat{\mathcal{E}}[\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}_a)\}/f_i \mid \forall\ [\![f_i]\!] \in \mathbf{Fn}], \\ & \perp[\{\lambda(v)\ .\ \mathcal{E}[\![e^{sp}]\!](\perp[v/x], \phi'_a)\}/f^{sp} \mid \forall\ \textit{specialized function}\ f^{sp}]\rangle.\end{aligned}$$

In this proof, i ranges over all user-defined functions.

Let $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ be the predicate over Φ such that:

$$\begin{aligned}\mathcal{R}^{\widehat{\mathcal{E}}_1}\Phi\ &=\ \mathcal{R}^{\widehat{\mathcal{E}}_1}\langle \phi, \hat{\phi}, \phi' \rangle \\ &=\ \forall\ [\![f_i]\!] \in \mathbf{Fn},\ \forall v_1, v_2 \in \mathbf{Values},\ \forall \hat{v}_1, \hat{v}_2 \in Result_{\widehat{\mathcal{E}}},\ \forall \rho_d \in VarEnv, \\ &\quad \bigwedge_{j=1}^{2} (\rho_d\ satisfies\ \langle v_j, \hat{v}_j \rangle)\ \Rightarrow\ \phi[\![f_i]\!](v_1, v_2) =_\perp \mathcal{E}[\![(\hat{\phi}[\![f_i]\!](\hat{v}_1, \hat{v}_2))\downarrow 1]\!](\rho_d, \phi').\end{aligned}$$

The predicate can be proved using Kleene's approximation over Φ, with the least element

$$\begin{aligned}\langle \phi_0, \hat{\phi}_0, \phi'_0 \rangle \;=\; \langle\; & \bot[(strict\ (\lambda(v_1, v_2)\ .\ \bot_{Values}))/f_i \mid \forall\ [\![f_i]\!] \in \mathbf{Fn}], \\ & \bot[(strict\ (\lambda(\hat{v}_1, \hat{v}_2)\ .\ \bot_{Result_{\widehat{\mathcal{E}}}}))/f_i \mid \forall\ [\![f_i]\!] \in \mathbf{Fn}], \\ & \bot[(strict\ (\lambda(v_1, v_2)\ .\ \bot_{Values})/f^{sp} \mid \forall\ specialized\ function\ f^{sp}]\rangle,\end{aligned}$$

and the predicate $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ over the $n+1^{st}$ approximation being

$$\begin{aligned}\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1} \equiv_{def}\ & \mathcal{R}^{\widehat{\mathcal{E}}_1}\langle \phi_{n+1}, \hat{\phi}_{n+1}, \phi'_{n+1}\rangle \\ =\ & \forall\ [\![f_i]\!] \in \mathbf{Fn},\ \forall v_1, v_2 \in \mathbf{Values},\ \forall \hat{v}_1, \hat{v}_2 \in Result_{\widehat{\mathcal{E}}},\ \forall \rho_d \in VarEnv, \\ & \bigwedge_{j=1}^{2} \rho_d\ satisfies\ \langle v_j, \hat{v}_j\rangle\ \Rightarrow \\ & \phi_{n+1}[\![f_i]\!](v_1, v_2) =_{\bot} \mathcal{E}[\![(\hat{\phi}_{n+1}[\![f_i]\!](\hat{v}_1, \hat{v}_2))\downarrow 1]\!](\rho_d, \phi'_{n+1}) \\ =\ & \forall\ [\![f_i]\!] \in \mathbf{Fn},\ \forall v_1, v_2 \in \mathbf{Values},\ \forall \hat{v}_1, \hat{v}_2 \in Result_{\widehat{\mathcal{E}}},\ \forall \rho_d \in VarEnv, \\ & \bigwedge_{j=1}^{2} \rho_d\ satisfies\ \langle v_j, \hat{v}_j\rangle\ \Rightarrow \\ & \mathcal{E}[\![e_i]\!](\bot[v_k/x_k], \phi_n) =_{\bot} \mathcal{E}[\![(\widehat{\mathcal{E}}[\![e_i]\!](\bot[\hat{v}_k/x_k], \hat{\phi}_n))\downarrow 1]\!](\rho_d, \phi'_{n+1}).\end{aligned}$$

Notice that at any $i+1^{st}$ approximation, ϕ'_{i+1} is obtained from the residual program produced by $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{E}}$, both having $\hat{\phi}_{i+1}$ as their function environment. Formally,

$$\phi'_{i+1} \;=\; \bot[(strict\{\lambda\ v\ .\ \mathcal{E}[\![e^{sp}]\!](\bot[v/x], \phi'_i)\})/f^{sp}]$$

for all specialized function f^{sp} with body e^{sp}.

ϕ' is derived from cache $\hat{\sigma}$ produced by $\widehat{\mathcal{A}}$ and ϕ'_i is derived from cache $\hat{\sigma}_i$ at the i^{th} approximation. Below are properties about $\hat{\sigma}_i$ and ϕ'_i.

Property 3.2 $\forall\ i \in \{0, 1, \ldots\},\ \hat{\sigma}_i \sqsubseteq_{Cache} \hat{\sigma}_{i+1}$.

Proof : From the result that $\hat{\sigma}_i$'s are the cache produced by $\widehat{\mathcal{A}}$ with function environment $\hat{\phi}_i$ and $\widehat{\mathcal{A}}$ is continuous over all its arguments. □

Property 3.3 $\forall\ i \in \{0, 1, \ldots\},\ \phi'_i \sqsubseteq_{FunEnv} \phi'_{i+1}$.

Proof : Since $\forall i \in \{0, 1, \ldots\}$, ϕ'_i is obtained from the residual program, which is the result of $\widehat{\mathcal{E}}_{Prog}$. Inspecting the function definition of $\widehat{\mathcal{E}}_{Prog}$ shows that it is continuous over all its arguments. In particular, since $\forall\ i \in \{0, 1, \ldots\},\ \hat{\sigma}_i \sqsubseteq_{Cache} \hat{\sigma}_{i+1}$, therefore $\phi'_i \sqsubseteq_{FunEnv} \phi'_{i+1}$. □

We prove the validity of $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ by fixed-point induction:

For the least element, $\langle \phi_0, \hat{\phi}_0, \phi'_0 \rangle$, we have $\phi[\![f_i]\!](v_1, v_2) = \perp_{Values}$ and $\hat{\phi}[\![f_i]\!](\hat{v}_1, \hat{v}_2) = \perp_{Result_{\widehat{\mathcal{E}}}}$. Thus, $\mathcal{R}^{\widehat{\mathcal{E}}_1}\langle \phi_0, \hat{\phi}_0, \phi'_0 \rangle$ holds vacuously.

Suppose that $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ is true for some element $\langle \phi_n, \hat{\phi}_n, \phi'_n \rangle$ in the ascending chain, we want to prove that $\mathcal{R}^{\widehat{\mathcal{E}}_1}$ is true for $\langle \phi_{n+1}, \hat{\phi}_{n+1}, \phi'_{n+1} \rangle = \Phi\langle \phi_n, \hat{\phi}_n, \phi'_n \rangle$.

For clarity, we introduce the following abbreviations:

1. $\perp[v_k/x_k]$ is abbreviated by ρ and $\perp[\hat{v}_k/x_k]$ by $\hat{\rho}$.

2. Given an expression e, we abbreviate $\mathcal{E}[\![e]\!](\rho, \phi_n)$ by $[\![e]\!]_{\mathcal{E}}$, and $\widehat{\mathcal{E}}[\![e]\!](\hat{\rho}, \hat{\phi}_n)$ by $[\![e]\!]_{\widehat{\mathcal{E}}}$.

The proof of $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ requires structural induction on e.

- If e is a constant or a variable, the proof is trivial, and thus omitted.

- e is a primitive call, $[\![p(e_1, \ldots, e_n)]\!]$. Let $v = [\![p(e_1, \ldots, e_n)]\!]_{\mathcal{E}}$, $\hat{v} = [\![p(e_1, \ldots, e_n)]\!]_{\widehat{\mathcal{E}}}$.

  - $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds trivially if $\hat{v} = \perp_{Result_{\widehat{\mathcal{E}}}}$.
  - If $\hat{v}\downarrow 1$ is a constant, then $\hat{v}\downarrow 1 = \hat{\tau}(v)$ from Theorem 3.3. Therefore, $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds in this case.
  - If $\hat{v} \downarrow 1$ is not a constant, then the residual expression is of the form $[\![p(e''_1, \ldots, e''_n)]\!]$, where $e''_i = [\![e_i]\!]_{\widehat{\mathcal{E}}}$ $\forall i \in \{1, \ldots, n\}$. By the structural-induction hypothesis, $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds for all the arguments of the primitive call. Furthermore, since ρ and $\hat{\rho}$ contain all the bindings for free variables in e, they also contain the bindings for free variables of the arguments. We thus have:

$$\begin{aligned}
&\mathcal{E}[\![[\![p(e_1,\ldots,e_n)]\!]_{\widehat{\mathcal{E}}}]\!](\rho_d,\phi'_{n+1})\\
&= \mathcal{E}[\![p(e''_1,\ldots,e''_n)]\!](\rho_d,\phi'_{n+1})\\
&= \mathcal{K}_P[\![p]\!]((\mathcal{E}[\![e''_1]\!](\rho_d,\phi'_{n+1})),\ldots,(\mathcal{E}[\![e''_n]\!](\rho_d,\phi'_{n+1})))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[from standard semantics]}\\
&=_\perp \mathcal{K}_P[\![p]\!]([\![e_1]\!]_\mathcal{E},\ldots,[\![e_n]\!]_\mathcal{E}) \qquad \text{[structural-induction hypothesis]}\\
&= [\![p(e_1,\ldots,e_n)]\!]_\mathcal{E} \qquad \text{[from standard semantics]}
\end{aligned}$$

Therefore, $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds.

- e is a conditional expression, $[\![if\ e_1\ e_2\ e_3]\!]$. $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds trivially if e_1 is partially evaluated to $\perp_{Result_{\widehat{\mathcal{E}}}}$. If e_1 is partially evaluated to a constant, then the result of partially evaluating e is obtained from partially evaluating either e_2 or e_3. By the structural-induction hypothesis, $\mathcal{R}_{n+1}$ holds.

  If e_1 is partially evaluated to a residual expression, then the result of partially evaluating e has the form $[\![if\ e''_1\ e''_2\ e''_3]\!]$, where $e''_i = [\![e_i]\!]_{\widehat{\mathcal{E}}}\ \forall i \in \{1,\ldots,3\}$. Therefore,

$$\begin{aligned}
&\mathcal{E}[\![[\![if\ e_1\ e_2\ e_3]\!]_{\widehat{\mathcal{E}}}]\!](\rho_d,\phi'_{n+1})\\
&= \mathcal{E}[\![if\ e''_1\ e''_2\ e''_3]\!](\rho_d,\phi'_{n+1})\\
&= (\mathcal{E}[\![e''_1]\!](\rho_d,\phi'_{n+1})) \to (\mathcal{E}[\![e''_2]\!](\rho_d,\phi'_{n+1})),(\mathcal{E}[\![e''_3]\!](\rho_d,\phi'_{n+1}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[from standard semantics]}\\
&=_\perp [\![e_1]\!]_\mathcal{E} \to [\![e_2]\!]_\mathcal{E},[\![e_3]\!]_\mathcal{E} \qquad \text{[structural-induction hypothesis]}\\
&= [\![if\ e_1\ e_2\ e_3]\!]_\mathcal{E} \qquad \text{[from standard semantics]}
\end{aligned}$$

  Thus, $\mathcal{R}^{\widehat{\mathcal{E}}_1}_{n+1}$ holds.

- e is a function application, $[\![f_i(e_1,e_2)]\!]$. Partially evaluating e may result in the application being either unfolded or specialized. Suppose that the application is specialized, without loss of generality, we assume that the first argument of the application is static and propagated, whereas the second argument is dynamic. Then $[\![f_i(e_1,e_2)]\!]_{\widehat{\mathcal{E}}}$ becomes $[\![f_i^{sp}([\![e_2]\!]_{\widehat{\mathcal{E}}})]\!]$ where f_i^{sp} is the specialized function.

The partial-evaluation signature obtained from this application is (by the definition of $\widehat{\mathcal{A}}$) $\langle \mathrm{s}, [\![e_1]\!]_{\widehat{\mathcal{E}}}, \langle [\![x_2]\!], \hat{\delta}' \rangle \rangle$ where

$$\begin{aligned}
\hat{\delta}' &= \langle \top_{\widehat{Values}}, \hat{\delta}^2, \ldots, \hat{\delta}^m \rangle, \\
\langle [\![e_1]\!]_{\widehat{\mathcal{E}}}, \langle [\![x_2]\!], \hat{\delta}' \rangle \rangle &= SpPat([\![f_i]\!], \langle [\![e_1]\!]_{\widehat{\mathcal{E}}}, [\![e_2]\!]_{\widehat{\mathcal{E}}} \rangle, \langle static, dynamic \rangle), \\
\langle -, \langle \hat{\delta}^1, \hat{\delta}^2, \ldots, \hat{\delta}^n \rangle \rangle &= [\![e_2]\!]_{\widehat{\mathcal{E}}}.
\end{aligned}$$

Thus, the specialized function f_i^{sp} is included in the residual program produced; its definition is as follows.

$$\begin{aligned}
f_i^{sp}(x_2) &= [\![(\hat{\phi}_n[\![f_i]\!](\langle [\![e_1]\!]_{\widehat{\mathcal{E}}}, \langle [\![x_2]\!], \hat{\delta}' \rangle))\downarrow 1]\!] \\
&= [\![(\hat{\phi}_n[\![f_i]\!](\hat{\tau}([\![e_1]\!]\varepsilon), \langle [\![x_2]\!], \hat{\delta}' \rangle))\downarrow 1]\!]
\end{aligned}$$

The last equality holds by structural-induction hypothesis and by the fact that only constants are allowed to be propagated in a function specialization. The corresponding entry of f_i^{sp} in ϕ'_{n+1} is

$$strict(\lambda v \,.\, \mathcal{E}[\![(\hat{\phi}_n[\![f_i]\!](\hat{\tau}([\![e_1]\!]\varepsilon), \langle [\![x_2]\!], \hat{\delta}' \rangle))\downarrow 1]\!](\bot[v/x_2], \phi'_n)) \qquad (3.2)$$

Thus, we have

$$\begin{aligned}
&\mathcal{E}[\![f_i^{sp}([\![e_2]\!]_{\widehat{\mathcal{E}}})]\!](\rho_d, \phi'_{n+1}) \\
&= \phi'_{n+1}[\![f_i^{sp}]\!](\mathcal{E}[\![e_2]\!]_{\widehat{\mathcal{E}}}(\rho_d, \phi'_{n+1})) \\
&=_{\bot} \phi'_{n+1}[\![f_i^{sp}]\!]([\![e_2]\!]\varepsilon) && \text{[structural-induction hypothesis]} \\
&= \mathcal{E}[\![(\hat{\phi}_n[\![f_i]\!](\hat{\tau}([\![e_1]\!]\varepsilon), \langle [\![x_2]\!], \hat{\delta}' \rangle))\downarrow 1]\!](\bot[[\![e_2]\!]\varepsilon/x_2], \phi'_n) \\
&=_{\bot} \phi_n[\![f_i]\!]([\![e_1]\!]\varepsilon, [\![e_2]\!]\varepsilon) && \text{[fixed-point-induction hypothesis]} \\
&= [\![f_i(e_1, e_2)]\!]\varepsilon
\end{aligned}$$

In the derivation above, the fourth equality is valid based on an instance of our fixed-point-induction hypothesis. This is because $(\bot[[\![e_2]\!]\varepsilon/x_2])$ is the only environment that satisfies both the pairs $\langle [\![e_1]\!]\varepsilon, \hat{\tau}([\![e_1]\!]\varepsilon) \rangle$ and $\langle [\![e_2]\!]\varepsilon, \langle [\![x_2]\!], \hat{\delta}' \rangle \rangle$. Therefore, $\mathcal{R}_{n+1}^{\widehat{\varepsilon_1}}$ holds for the application.

On the other hand, consider the case where the application is unfolded. The equality in $\mathcal{R}_{n+1}^{\widehat{\varepsilon_1}}$ becomes

$$\phi_n[\![f_i]\!]([\![e_1]\!]\varepsilon, [\![e_2]\!]\varepsilon) =_{\bot} \mathcal{E}[\![(\hat{\phi}_n[\![f_i]\!]([\![e_1]\!]_{\widehat{\mathcal{E}}}, [\![e_2]\!]_{\widehat{\mathcal{E}}}))\downarrow 1]\!]\ (\rho_d,\ \phi'_{n+1}). \qquad (3.3)$$

For Equation 3.3 to hold, ρ_d must satisfy both $\langle [\![e_1]\!]\varepsilon, [\![e_1]\!]_{\widehat{\varepsilon}}\rangle$ and $\langle [\![e_2]\!]\varepsilon, [\![e_2]\!]_{\widehat{\varepsilon}}\rangle$. That is, $\forall i \in \{1,2\}$,

$$(v_i =_\perp \mathcal{E}[\![\hat{v}_i{\downarrow}1]\!](\rho_d, \phi'_{n+1})) \wedge (v_i \preceq_{\hat{\alpha}_{\widehat{SD}}} \hat{v}_i{\downarrow}2).$$

This is true by the structural-induction hypothesis, Property 3.3 about ϕ'_{n+1}, and Theorem 3.3.

Using ρ_d, the fixed-point-induction hypothesis is

$$\phi_n[\![f_i]\!]([\![e_1]\!]\varepsilon, [\![e_2]\!]\varepsilon) =_\perp \mathcal{E}[\![(\hat{\phi}_n[\![f_i]\!]([\![e_1]\!]_{\widehat{\varepsilon}}, [\![e_2]\!]_{\widehat{\varepsilon}})){\downarrow}1]\!](\rho_d, \phi'_n),$$

Notice that the only difference between the hypothesis and Equation 3.3 is the usage of ϕ'_n and ϕ'_{n+1}. Let $e' = (\hat{\phi}_n[\![f_i]\!](([\![e_1]\!]_{\widehat{\varepsilon}}), ([\![e_2]\!]_{\widehat{\varepsilon}}))){\downarrow}1$. Since the domain **Exp** is flat, the only case where Equation 3.3 may have failed to hold would be when standard evaluation of e' made references to specialized functions defined in ϕ'_{n+1}. Suppose that f^{sp} were such a function, and its call in e' were $[\![f^{sp}(r'_2)]\!]$. This residual call would be the result of partially evaluating a function call. Let the function call be $[\![f(r_1, r_2)]\!]$. Then, it would be the case that at the n^{th} approximation, we had

$$[\![f(r_1, r_2)]\!]\varepsilon =_\perp \mathcal{E}[\![[\![f(r_1, r_2)]\!]_{\widehat{\varepsilon}}]\!](\rho_d, \phi'_n) = \mathcal{E}[\![f^{sp}(r'_k)]\!](\rho_d, \phi'_n)$$

be true vacuously (by the hypothesis), but at the $n+1^{st}$ approximation, the equation

$$[\![f(r_1, r_2)]\!]\varepsilon =_\perp \mathcal{E}[\![f^{sp}(r'_k)]\!](\rho_d, \phi'_{n+1}) \tag{3.4}$$

became false. However, Equation 3.4 is the result of function specialization, and we have already proved its validity. Thus, we arrive at a contradiction, and Equation 3.3 must therefore hold. Hence, $\mathcal{R}^{\widehat{\varepsilon}_1}_{n+1}$ holds.

Hence, $\mathcal{R}^{\widehat{\varepsilon}_1}\langle \phi, \hat{\phi}, \phi' \rangle$ holds. This concludes the proof. □

Correctness of $\mathcal{R}^{\widehat{\mathcal{E}}}$

Now, we are ready to define the relation between $\mathcal{E}$ and $\widehat{\mathcal{E}}$. This is defined in terms of the result of Theorem 3.3 and Lemma 3.6. Firstly, since both $\mathcal{E}$ and $\widehat{\mathcal{E}}$ take variable environments as their arguments, we need to relate these environments. To do so, we extend the notion of satisfiability to define the relationship between variable environments, instead of pairs of related values. This is a variant of the notion of *agreeability* as defined by Gomard in [Gomard, 1992].

Definition 3.5 (Agreeability) *Let P be a program in our first-order language. Suppose ϕ' is the function environment, defined by the standard semantics, for a specialized version of P. Also, let $\rho, \rho_d \in VarEnv$ be two variable environments defined by the standard semantics, and $\hat{\rho} \in \widehat{VarEnv}$ be a variable environment defined by the partial-evaluation semantics. For any expression, e in P, ρ, $\hat{\rho}$ and ρ_d* agree on e at ϕ' *if*

$$\forall [\![x]\!] \in FV(e),\ \rho[\![x]\!] =_{\perp} \mathcal{E}[\![(\hat{\rho}[\![x]\!])\downarrow 1]\!](\rho_d, \phi') \ \wedge\ \rho[\![x]\!] \preceq_{\hat{a}_{\widehat{SD}}} (\hat{\rho}[\![x]\!])\downarrow 2.$$

The notion of satisfiability can then be expressed in terms of agreeability as follows.

Observation 3.2 *Given that ρ_d satisfies all the pairs in the set $\{\langle v_1, \hat{v}_1\rangle, \ldots, \langle v_n, \hat{v}_n\rangle\}$. Let $\rho = \perp[v_1/x_1, \ldots, v_n/x_n]$, and $\hat{\rho} = \perp[\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n]$. Then, for any expression e in P with $FV(e) = \{x_1, \ldots, x_n\}$, we must have ρ, $\hat{\rho}$ and ρ_d agree on e.*

Notice that ρ and $\hat{\rho}$ as defined in Observation 3.2 represent how all the variable environments used in standard and partial-evaluation semantics are constructed. Therefore, the result of Lemma 3.6 can be expressed in terms of an arbitrary expression in program P as follows.

Corollary 3.1 *Given a program P in our first-order language. Let ϕ and $\hat{\phi}$ be the two function environments for P defined by the standard and the partial-evaluation*

semantics respectively. Let ϕ' be the function environment, defined by the standard semantics, for a a specialized version of program P. Then, for any expression e in P, $\forall \rho, \hat{\rho} \in VarEnv$ and $\rho_d \in \widehat{VarEnv}$ that agree on e at ϕ', we have

$$\mathcal{E}[\![e]\!](\rho, \phi) =_{\perp} \mathcal{E}\,[\![(\hat{\mathcal{E}}[\![e]\!](\hat{\rho}, \hat{\phi}))\downarrow 1]\!](\rho_d, \phi').$$

Correctness of the local partial-evaluation semantics can be stated as follows:

Theorem 3.5 (Correctness of Local Partial-Evaluation Semantics) *Given a program P in our first-order language. Let ϕ and $\hat{\phi}$ be the two function environments for P defined by the standard and the partial-evaluation semantics respectively. Let ϕ' be the function environment, defined by the standard semantics, for a specialized version of program P. Then, for any expression e in P, $\forall\ \rho, \hat{\rho} \in VarEnv$ and $\rho_d \in \widehat{VarEnv}$ that agree on e at ϕ', we have*

$$\begin{aligned} \mathcal{E}[\![e]\!](\rho, \phi) &=_{\perp} \mathcal{E}\,[\![(\hat{\mathcal{E}}[\![e]\!](\hat{\rho}, \hat{\phi}))\downarrow 1]\!](\rho_d, \phi') \quad \textit{and} \\ \mathcal{E}[\![e]\!](\rho, \phi) &\preceq_{\hat{\alpha}_{\widehat{SD}}} (\hat{\mathcal{E}}[\![e]\!](\hat{\rho}, \hat{\phi}))\downarrow 2. \end{aligned}$$

Proof : From Theorem 3.3 and Corollary 3.1. □

3.5 Off-Line PPE

Off-line PPE of a program consists of two phases: the preprocessing phase called *facet analysis*, and the *specialization* phase. In this section, we use the technique of logical relation to define and prove the correctness of facet analysis. This conforms to our intuition that facet analysis is an abstraction of on-line PPE. Lastly, we describe a systematic way of deriving the specializer from on-line partial evaluation using the result of facet analysis.

- Semantic Domains

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

$$\tilde{\rho} \in \widetilde{\mathit{VarEnv}} = \mathbf{Var} \rightarrow \widetilde{\mathcal{SD}}$$

$$\tilde{\phi} \in \widetilde{\mathit{FunEnv}} = \mathbf{Fn} \rightarrow \widetilde{\mathcal{SD}}^n \rightarrow \widetilde{\mathcal{SD}}$$

$$\widetilde{\mathit{Env}} = \widetilde{\mathit{VarEnv}} \times \widetilde{\mathit{FunEnv}}$$

$$\tilde{s} \in \mathbf{Sig} = (\mathbf{Transf} \times \widetilde{\mathcal{SD}}^n)$$

$$\tilde{\sigma} \in \mathit{Result}_{\tilde{\mathcal{A}}} = \mathbf{Cache}_{\tilde{\mathcal{A}}} = \mathbb{F}\mathbf{n} \rightarrow \mathbf{Sig}$$

- Valuation Functions

$$\tilde{\mathcal{E}}_{\mathit{Prog}} : \mathbf{Program} \rightarrow \widetilde{\mathcal{SD}}^n \rightarrow \mathbf{Cache}_{\tilde{\mathcal{A}}}$$

$$\tilde{\mathcal{E}}_{\mathit{Prog}}\ [\![\{f_i(x_1,\ldots,x_n) = e_i\}]\!]\ \langle \tilde{\delta}_1,\ldots,\tilde{\delta}_n\rangle = \tilde{h}(\bot[\langle \mathbf{s}, \tilde{\delta}_1,\ldots,\tilde{\delta}_n\rangle / f_1])$$

$$\textit{whererec}\quad \tilde{h}(\tilde{\sigma}) = \tilde{\sigma} \sqcup \tilde{h}(\bigsqcup\{\tilde{\mathcal{A}}\ [\![e_i]\!]\ (\bot[\tilde{\delta}_k/x_k], \tilde{\phi}) \mid \langle -, \tilde{\delta}_1,\ldots,\tilde{\delta}_n\rangle = \tilde{\sigma}[\![f_i]\!], [\![f_i]\!] \in \mathit{Dom}(\tilde{\sigma})\})$$

$$\tilde{\phi} = \bot[\{\lambda(\tilde{\delta}_1,\ldots,\tilde{\delta}_n)\ .\ \tilde{\mathcal{E}}\ [\![e_i]\!](\bot[\tilde{\delta}_k/x_k], \tilde{\phi})\}/f_i]$$

$$\tilde{\mathcal{E}} = \overline{\mathcal{E}}$$

$$\tilde{\mathcal{A}} = \overline{\mathcal{A}}$$

Figure 3.11: Facet Analysis — Domains and Main functions

3.5.1 Specification of Facet Analysis

Analogous to the definition of on-line PPE, we assume the binding-time facet to be always defined in facet analysis. The main semantic domain used by the analysis is noted $\widetilde{\mathcal{SD}}$. It is a sum of products of abstract facets. The binding-time facet is assigned to the first component of each product. For brevity, we write $\top_{\widetilde{\mathcal{SD}}}$ to represent the maximum value of any summand of $\widetilde{\mathcal{SD}}$.

Figures 3.11 to 3.12 display the facet analysis for our first-order language. 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 in domain **Sig** is created when a function call is processed by facet analysis. It consists of two components: A transformation tag similar to that used in the partial-evaluation

- Local Combinator Definitions

$$Const_{\widetilde{\mathcal{E}}}\ [\![c]\!] = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \widetilde{\mathcal{K}}\ [\![c]\!]$$

$$VarLookup_{\widetilde{\mathcal{E}}}\ [\![x]\!] = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \tilde{\rho}[\![x]\!]$$

$$PrimOp_{\widetilde{\mathcal{E}}}\ [\![p]\!]\ (\tilde{k}_1,\ldots,\tilde{k}_n) = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \widetilde{\mathcal{K}}_P[\![p]\!]\ (\tilde{k}_1(\tilde{\rho},\tilde{\phi}),\ldots,\tilde{k}_n(\tilde{\rho},\tilde{\phi}))$$

$$Cond_{\widetilde{\mathcal{E}}}\ (\tilde{k}_1,\tilde{k}_2,\tilde{k}_3) = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \tilde{\delta}_1 = \bot_{\widetilde{\mathcal{SD}}} \rightarrow \bot_{\widetilde{\mathcal{SD}}},\ \tilde{\delta}_1^1 = static \rightarrow \tilde{\delta}_2 \sqcup \tilde{\delta}_3,\ \top_{\widetilde{\mathcal{SD}}}$$

$$\text{where } \tilde{\delta}_i = \tilde{k}_i(\tilde{\rho},\tilde{\phi}) \quad \forall i \in \{1,2,3\}$$

$$App_{\widetilde{\mathcal{E}}}\ [\![f]\!]\ (\tilde{k}_1,\ldots,\tilde{k}_n) = \lambda(\tilde{\rho},\tilde{\phi})\ .\ (Ft[\![f]\!])\downarrow 1\ (\tilde{\delta}_1^1,\ldots,\tilde{\delta}_n^1) = \mathrm{u} \rightarrow \tilde{\phi}\ [\![f]\!]\ (\tilde{\delta}_1,\ldots,\tilde{\delta}_n),\ \top_{\widetilde{\mathcal{SD}}}$$

$$\text{where } \tilde{\delta}_i = \tilde{k}_i(\tilde{\rho},\tilde{\phi}) \quad \forall i \in \{1,\ldots,n\}$$

- Primitive Functions

$$\widetilde{\mathcal{K}} : \mathbf{Const} \rightarrow \widetilde{\mathcal{SD}}$$

$$\widetilde{\mathcal{K}}\ [\![c]\!] = \langle \tilde{\Gamma}^1(d),\ldots,\tilde{\Gamma}^m(d)\rangle \text{ where } \tilde{\Gamma}^i = \tilde{\alpha}_{\widetilde{\mathcal{D}}^i} \circ \hat{\alpha}_{\widehat{\mathcal{D}}^i} \text{ and } d = \mathcal{K}\ [\![c]\!]$$

$$\widetilde{\mathcal{K}}_P : \mathbf{Po} \rightarrow \widetilde{\mathcal{SD}}^n \rightarrow \widetilde{\mathcal{SD}}$$

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

$$\widetilde{\mathcal{K}}_P\ [\![p^o]\!]\ (\tilde{\delta}_1,\ldots,\tilde{\delta}_n) = \tilde{b} = \bot_{\widetilde{Values}} \rightarrow \bot_{\widetilde{\mathcal{SD}}}, \langle \tilde{b}, \top_{\widetilde{\mathcal{D}}'^2},\ldots,\top_{\widetilde{\mathcal{D}}'^m}\rangle$$

$$\text{where } p^o : \mathbf{D}^n \rightarrow \mathbf{D}'$$

$$\tilde{b} = \tilde{\omega}_{p^o}(\tilde{\delta}_1,\ldots,\tilde{\delta}_n)$$

- Global Combinator Definitions

$$Const_{\widetilde{\mathcal{A}}}\ [\![c]\!] = \lambda(\tilde{\rho},\tilde{\phi})\ .\ (\lambda f\ .\ \bot_{Sig})$$

$$VarLookup_{\widetilde{\mathcal{A}}}\ [\![x]\!] = \lambda(\tilde{\rho},\tilde{\phi})\ .\ (\lambda f\ .\ \bot_{Sig})$$

$$PrimOp_{\widetilde{\mathcal{A}}}\ [\![p]\!]\ (\tilde{a}_1,\ldots,\tilde{a}_n) = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \bigsqcup_{i=1}^{n} \tilde{a}_i(\tilde{\rho},\tilde{\phi})$$

$$Cond_{\widetilde{\mathcal{A}}}\ (\tilde{a}_1,\tilde{a}_2,\tilde{a}_3)\ \tilde{k}_1 = \lambda(\tilde{\rho},\tilde{\phi})\ .\ \tilde{a}_1(\tilde{\rho},\tilde{\phi}) \sqcup \tilde{a}_2(\tilde{\rho},\tilde{\phi}) \sqcup \tilde{a}_3(\tilde{\rho},\tilde{\phi})$$

$$App_{\widetilde{\mathcal{A}}}\ [\![f]\!]\ (\tilde{a}_1,\ldots,\tilde{a}_n)\ (\tilde{k}_1,\ldots,\tilde{k}_n) = \lambda(\tilde{\rho},\tilde{\phi})\ .\ (\bigsqcup_{i=1}^{n} \tilde{a}_i(\tilde{\rho},\tilde{\phi})) \sqcup \tilde{\sigma}'$$

$$\text{where } \tilde{\sigma}' = (Ft[\![f]\!])\downarrow 1\ (\tilde{\delta}_1^1,\ldots,\tilde{\delta}_n^1) = \mathrm{u} \rightarrow \bot[\langle \mathrm{u},\tilde{\delta}_1,\ldots,\tilde{\delta}_n\rangle/f],\ \bot[\langle \mathrm{s},\tilde{\delta}_1',\ldots,\tilde{\delta}_n'\rangle/f]$$

$$\tilde{\delta}_i' = \langle b_i,\tilde{\delta}_i^2,\ldots,\tilde{\delta}_i^m\rangle \quad \forall i \in \{1,\ldots,n\}$$

$$\langle b_1,\ldots,b_n\rangle = (Ft[\![f]\!])\downarrow 2\ (\tilde{\delta}_1^1,\ldots,\tilde{\delta}_n^1)$$

$$\tilde{\delta}_i = \tilde{k}_i(\tilde{\rho},\tilde{\phi}) \quad \forall i \in \{1,\ldots,n\}$$

Figure 3.12: Facet Analysis — Local and Global semantic functions

signature, and the result of filter computation on call arguments.

The valuation function $\tilde{\mathcal{E}}$ is used to define an abstract version of each user-defined function. The resulting abstract functions are then used by the valuation function $\tilde{\mathcal{A}}$ to compute facet signatures. These signatures are recorded in a cache (from domain $\mathbf{Cache}_{\tilde{\mathcal{A}}}$). As usual, computation is accomplished via fixed-point iteration. Functions $\tilde{\mathcal{K}}$ and $\tilde{\mathcal{K}}_P$ perform the abstract computation on constants and primitive operators respectively.

The analysis is *monovariant*: each user-defined function is associated with *one* facet signature. Various facet signatures associated with a function at different call sites are folded into one signature using the lub operation. This operation is defined as follows:

$$\begin{aligned}
&\forall \tilde{\sigma}_1, \tilde{\sigma}_2 \in \mathbf{Cache}_{\tilde{\mathcal{A}}},\ \tilde{\sigma}_1 \sqcup \tilde{\sigma}_2 \ = \ \bot[\langle t, \tilde{\delta}_1, \ldots, \tilde{\delta}_n\rangle / f \mid \forall [\![f]\!] \in Dom(\tilde{\sigma}_1) \cup Dom(\tilde{\sigma}_2)] \\
&\quad \textit{where}\ \langle t, \tilde{\delta}_1, \ldots, \tilde{\delta}_n\rangle \ = \ ([\![f]\!] \in (Dom(\tilde{\sigma}_1) \cap Dom(\tilde{\sigma}_2))) \ \rightarrow \\
&\qquad\qquad \langle t' \sqcup t'',\ \tilde{\delta}'_1 \sqcup \tilde{\delta}''_1, \ldots, \tilde{\delta}'_n \sqcup \tilde{\delta}''_n\rangle, \\
&\qquad\qquad [\![f]\!] \in Dom(\tilde{\sigma}_1) \ \rightarrow \ \tilde{\sigma}_1[\![f]\!],\ \tilde{\sigma}_2[\![f]\!] \\
&\qquad \langle t', \tilde{\delta}'_1, \ldots, \tilde{\delta}'_n\rangle \ = \ \tilde{\sigma}_1[\![f]\!] \\
&\qquad \langle t'', \tilde{\delta}''_1, \ldots, \tilde{\delta}''_n\rangle \ = \ \tilde{\sigma}_2[\![f]\!]
\end{aligned}$$

3.5.2 Correctness of Facet Analysis

The initial input to facet analysis is an abstraction of the initial input to on-line partial evaluation. The facet analysis is correct if its final cache ($\mathbf{Cache}_{\tilde{\mathcal{A}}}$) contains the abstraction to all the partial-evaluation signatures of the on-line partial evaluation. The correctness is shown by relating the local and global semantics to their respective counterpart in the on-line partial-evaluation semantics. That is, we define a logical relation $\mathcal{R}^{\tilde{\mathcal{E}}}$ that relates $\hat{\mathcal{E}}$ and $\tilde{\mathcal{E}}$, and a logical relation $\mathcal{R}^{\tilde{\mathcal{A}}}$ that relates $\hat{\mathcal{A}}$ and $\tilde{\mathcal{A}}$. We first show the correctness of the local semantics defined by $\tilde{\mathcal{E}}$, and then that of the global semantics defined by $\tilde{\mathcal{A}}$.

Correctness of $\tilde{\mathcal{E}}$

We begin by defining a logical relation between product of facets and product of abstract facets.

Definition 3.6 (Relation $\preceq_{\tilde{\alpha}_{\widetilde{SD}}}$) *For any value $\hat{\delta} \in \widehat{SD}$ and $\tilde{\delta} \in \widetilde{SD}$,*

$$\hat{\delta} \preceq_{\tilde{\alpha}_{\widetilde{SD}}} \tilde{\delta} \Leftrightarrow \forall i \in \{1,\ldots,m\},\ \hat{\delta}^i \preceq_{\tilde{\alpha}_{\widetilde{D}^i}} \tilde{\delta}^i.$$

where $\preceq_{\tilde{\alpha}_{\widetilde{D}^i}}$ is the logical relation induced from the facet mapping from $\widehat{\mathbf{D}}^i$ to $\widetilde{\mathbf{D}}^i$.

Since $(\hat{\delta} \preceq_{\tilde{\alpha}_{\widetilde{SD}}} \tilde{\delta}) \equiv (\bigwedge_{i=1}^{m}(\hat{\delta}^i \preceq_{\tilde{\alpha}_{\widetilde{D}^i}} \tilde{\delta}^i))$, $\preceq_{\tilde{\alpha}_{\widetilde{SD}}}$ is a logical relation between $\widehat{SD}$ and $\widetilde{SD}$. Intuitively, $\hat{\delta} \preceq_{\tilde{\alpha}_{\widetilde{SD}}} \tilde{\delta}$ implies that $\tilde{\delta}$ is a safe approximation of value obtained by applying abstraction functions to each facet of $\hat{\delta}$. We say that in this case $\tilde{\delta}$ is an abstraction of $\hat{\delta}$. Relation $\mathcal{R}^{\tilde{\mathcal{E}}}$ defined below extends this abstraction relation to various domains used by the local semantic function.

Definition 3.7 (Relation $\mathcal{R}^{\tilde{\mathcal{E}}}$) *$\mathcal{R}^{\tilde{\mathcal{E}}}$ is a logical relation between domains of $\hat{\mathcal{E}}$ and $\tilde{\mathcal{E}}$ defined by:*

$$\begin{aligned}
&\hat{v}\ \mathcal{R}^{\tilde{\mathcal{E}}}_{Result_{\widetilde{\mathcal{E}}}}\ \tilde{\delta} \Leftrightarrow \hat{v}{\downarrow}2 \preceq_{\tilde{\alpha}_{\widetilde{SD}}} \tilde{\delta}\\
&\hat{\rho}\ \mathcal{R}^{\tilde{\mathcal{E}}}_{VarEnv}\ \tilde{\rho} \Leftrightarrow \forall [\![x]\!] \in \mathbf{Var},\ \hat{\rho}[\![x]\!]\ \mathcal{R}^{\tilde{\mathcal{E}}}_{Result_{\widetilde{\mathcal{E}}}}\ \tilde{\rho}[\![x]\!]\\
&\hat{\phi}\ \mathcal{R}^{\tilde{\mathcal{E}}}_{FunEnv}\ \tilde{\phi} \Leftrightarrow \forall [\![f]\!] \in \mathbf{Fn}, \forall i \in \{1,\ldots,n\}, \forall \hat{v}_i \in Result_{\widehat{\mathcal{E}}}, \forall \tilde{\delta}_i \in \widetilde{SD},\\
&\qquad \bigwedge_{i=1}^{n}(\hat{v}_i\ \mathcal{R}^{\tilde{\mathcal{E}}}_{Result_{\widetilde{\mathcal{E}}}}\ \tilde{\delta}_i) \Rightarrow \hat{\phi}[\![f]\!](\hat{v}_1,\ldots,\hat{v}_n)\ \mathcal{R}^{\tilde{\mathcal{E}}}_{Result_{\widetilde{\mathcal{E}}}}\ \tilde{\phi}[\![f]\!](\tilde{\delta}_1,\ldots,\tilde{\delta}_n)\\
&\langle \hat{d}_1, \hat{d}_2\rangle\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_1\times D_2}\ \langle \tilde{d}_1, \tilde{d}_2\rangle \Leftrightarrow \hat{d}_1\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_1}\ \tilde{d}_1 \wedge \hat{d}_2\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_2}\ \tilde{d}_2\\
&\hat{f}\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_1\to D_2}\ \tilde{f} \Leftrightarrow \forall \hat{d} \in \widehat{D}_1, \forall \tilde{d} \in \widetilde{D}_1,\ \hat{d}\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_1}\ \tilde{d} \Rightarrow \hat{f}(\hat{d})\ \mathcal{R}^{\tilde{\mathcal{E}}}_{D_2}\ \tilde{f}(\tilde{d}).
\end{aligned}$$

Lemma 3.7 *Given a program P. Let $\hat{\phi}$ and $\tilde{\phi}$ be the two function environments for P defined by the partial-evaluation semantics and the facet analysis respectively. Then $\hat{\phi}\ \mathcal{R}^{\tilde{\mathcal{E}}}\ \tilde{\phi}$.*

Proof : The proof is similar to the proof for Lemma 3.4, and is thus omitted. □

We can now express the correctness of the local semantic function using the relation $\mathcal{R}^{\tilde{\mathcal{E}}}$. The following theorem states that: if the input to $\tilde{\mathcal{E}}$ is an abstraction of the input to $\hat{\mathcal{E}}$, then the result of computation by $\tilde{\mathcal{E}}$ is still an abstraction of that by $\hat{\mathcal{E}}$.

Theorem 3.6 (Correctness of Local Facet Analysis) $\hat{\mathcal{E}}\ \mathcal{R}^{\tilde{\mathcal{E}}}\ \tilde{\mathcal{E}}$.

Proof : From Lemma 3.7. □

An immediate consequence of Theorem 3.6 is that when value *static* is obtained from computation done by $\tilde{\mathcal{E}}$, the corresponding result obtained from computation done by $\hat{\mathcal{E}}$ is either a constant or a bottom. This is stated precisely as follows:

Corollary 3.2 *Given a program P. For any expression e in P, and* $\forall \hat{\rho} \in \widehat{VarEnv}$,

$$(\tilde{\mathcal{E}}\ [\![e]\!](\tilde{\rho},\tilde{\phi}))\downarrow 1 \ = \ static \ \Rightarrow \ (\hat{\mathcal{E}}\ [\![e]\!](\hat{\rho},\hat{\phi}))\downarrow 1 \in \mathbf{Const} \cup \{\perp_{Exp}\}$$

where both $\hat{\phi} \in \widehat{FunEnv}$ *and* $\tilde{\phi} \in \widetilde{FunEnv}$ *are fixed for the program, and* $\tilde{\rho} \in \widetilde{VarEnv}$ *is defined such that* $\hat{\rho}\ \mathcal{R}^{\tilde{\mathcal{E}}}\ \tilde{\rho}$.

This result will be fully utilized when we consider the off-line specializer.

Correctness of the Global Analysis

We prove the correctness of the global analysis (1) by relating the semantics of $\tilde{\mathcal{A}}$ with that of $\hat{\mathcal{A}}$ using logical relation $\mathcal{R}^{\tilde{\mathcal{A}}}$, and (2) by showing that all the non-trivial calls that are recorded by $\hat{\mathcal{A}}$ are captured in the cache produced by $\tilde{\mathcal{A}}$.

Definition 3.8 (Relation $\mathcal{R}^{\tilde{\mathcal{A}}}$) $\mathcal{R}^{\tilde{\mathcal{A}}}$ *is a logical relation between the domains of* $\hat{\mathcal{A}}$ *and* $\tilde{\mathcal{A}}$ *defined by extending relation* $\mathcal{R}^{\tilde{\mathcal{E}}}$ *to include the relation between* $\hat{\sigma}$ *and* $\tilde{\sigma}$ *produced by* $\hat{\mathcal{A}}$ *and* $\tilde{\mathcal{A}}$ *respectively:*

$$\langle \hat{t}, \hat{v}_1, \ldots, \hat{v}_n \rangle\ \mathcal{R}^{\tilde{\mathcal{A}}}_{Sig}\ \langle \tilde{t}, \tilde{\delta}_1, \ldots, \tilde{\delta}_n \rangle \ \Leftrightarrow \ (\hat{t} \sqsubseteq_{Transf} \tilde{t}) \ \wedge \ \bigwedge_{i=1}^{n} (\hat{v}_i\ \mathcal{R}^{\tilde{\mathcal{E}}}_{Result_{\tilde{\mathcal{E}}}}\ \tilde{\delta}_i)$$

$$\hat{\sigma}\ \mathcal{R}^{\tilde{\mathcal{A}}}_{Result_{\tilde{\mathcal{A}}}}\ \tilde{\sigma} \ \Leftrightarrow \ \forall\ [\![f]\!] \in Dom(\sigma), \forall \hat{s} \in \hat{\sigma}[\![f]\!], \exists \tilde{s} \in \tilde{\sigma}[\![f]\!] \text{ such that } \hat{s}\ \mathcal{R}^{\tilde{\mathcal{A}}}_{Sig}\ \tilde{s}$$

We note that $\mathcal{R}^{\tilde{\mathcal{A}}}$ is an extension of $\mathcal{R}^{\tilde{\mathcal{E}}}$. Therefore, we say that $\tilde{\sigma}$ is an abstraction of $\hat{\sigma}$ if $\hat{\sigma}\ \mathcal{R}^{\tilde{\mathcal{A}}}_{Result_{\tilde{\mathcal{A}}}}\ \tilde{\sigma}$. We also notice the lub operations defined on both caches preserve $\mathcal{R}^{\tilde{\mathcal{A}}}_{Result_{\tilde{\mathcal{A}}}}$. With this relation, the next lemma shows that all the partial-evaluation signatures recorded in the final cache produced by $\hat{\mathcal{A}}$ are captured in the corresponding cache produced by $\tilde{\mathcal{A}}$ in the sense the latter is an abstraction of the former.

Lemma 3.8 *Given a program P. Let $\hat{\phi}$ and $\tilde{\phi}$ be two function environments for P defined by the partial evaluation and the facet analysis respectively. For any expression e in P, for any $\hat{\rho}$, $\tilde{\rho}$ such that $\hat{\rho}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{\rho}$,*

$$\hat{\mathcal{A}}\ [\![e]\!](\hat{\rho},\hat{\phi})\quad \mathcal{R}^{\tilde{\mathcal{A}}}\quad \tilde{\mathcal{A}}\ [\![e]\!](\tilde{\rho},\tilde{\phi}).$$

Proof : The proof is by structural induction over an expression. Firstly, notice that $\hat{\phi}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{\phi}$. It then suffices to show that $\mathcal{R}^{\tilde{\mathcal{A}}}$ holds for all the corresponding pairs of combinators used by $\hat{\mathcal{A}}$ and $\tilde{\mathcal{A}}$ respectively. By structural induction, it is easy to see that $\mathcal{R}^{\tilde{\mathcal{A}}}$ holds for constant, variable and primitive calls. We show below that $\mathcal{R}^{\tilde{\mathcal{A}}}$ holds for the case of conditional expressions and function applications.

1. $Cond_{\tilde{\mathcal{A}}}$: By the structural-induction hypothesis, all the corresponding pairs of arguments are related by $\mathcal{R}^{\tilde{\mathcal{A}}}$. Since the result of $Cond_{\tilde{\mathcal{A}}}$ is the lub of the caches produced at all the arguments, whereas the result of $Cond_{\hat{\mathcal{A}}}$ is the lub of the caches produced at some of the arguments, $\mathcal{R}^{\tilde{\mathcal{A}}}$ must hold.

2. $App_{\tilde{\mathcal{A}}}$: By the structural-induction hypothesis, $\mathcal{R}^{\tilde{\mathcal{A}}}$ holds for all the arguments to the application.

   Let $\hat{\sigma} = \bot[\{\langle \hat{t}, \hat{v}''_1, \ldots, \hat{v}''_n\rangle\}/f]$ and $\tilde{\sigma} = \bot[\langle \tilde{t}, \tilde{\delta}''_1, \ldots, \tilde{\delta}''_n\rangle/f]$, we need to show that $\hat{\sigma}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{\sigma}$.

   We consider the cases with different transformation values produced at the facet analysis level.

- If $\tilde{t} = \mathbf{u}$, then $\hat{t} = \mathbf{u}$ by the monotonicity of filters. Thus, $\forall\, i \in \{1, \ldots, n\}$,

$$\begin{array}{lll} \hat{v}_i'' = \hat{a}_i(\hat{\rho}, \hat{\phi}_n) & & \text{[by definition]} \\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{a}_i(\tilde{\rho}, \tilde{\phi}_n) & \text{[structural-induction hypothesis]} & \\ = \tilde{\delta}_i''. & & \text{[by definition]} \end{array}$$

  Therefore, $\hat{\sigma}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{\sigma}$.

- If $\tilde{t} = \mathbf{s}$, then $\hat{t} \sqsubseteq_{Transf} \tilde{t}$ by monotonicity of the filter.
  Let $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ and $\langle \tilde{\delta}_1, \ldots, \tilde{\delta}_n \rangle$ be the initial arguments computed for the application. By structural induction, $\forall i \in \{1, \ldots, n\}, \hat{v}_i \mathcal{R}^{\tilde{\mathcal{A}}} \tilde{\delta}_i$. By the monotonicity of filter,

$$\langle \hat{b}_1, \ldots, \hat{b}_n \rangle \sqsubseteq \langle \tilde{b}_1, \ldots, \tilde{b}_n \rangle$$

  where $\langle \hat{b}_1, \ldots, \hat{b}_n \rangle = (Ft[\![f]\!])\downarrow 2\ (\hat{bt}(\hat{v}_1), \ldots, \hat{bt}(\hat{v}_n))$ *and*
  $\langle \tilde{b}_1, \ldots, \tilde{b}_n \rangle = (Ft[\![f]\!])\downarrow 2\ (\tilde{\delta}_1^1, \ldots, \tilde{\delta}_n^1)$
  Let $\langle \hat{v}_1'', \ldots, \hat{v}_n'' \rangle = SpPat\ ([\![f]\!], \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \langle \hat{b}_1, \ldots, \hat{b}_n \rangle)$. From the definition of *SpPat* and Property 3.1, we have

$$SpPat\ ([\![f]\!], \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \langle \hat{b}_1, \ldots, \hat{b}_n \rangle)\ \ \mathcal{R}^{\tilde{\mathcal{A}}}\ \ \langle \tilde{\delta}_1', \ldots, \tilde{\delta}_n' \rangle,$$

  where $\forall i \in \{1, \ldots, n\},\ \tilde{\delta}_i' = \langle \tilde{b}_i, \tilde{\delta}_i^2, \ldots, \tilde{\delta}_i^m \rangle$. Since $\langle \tilde{t}, \tilde{\delta}_1', \ldots, \tilde{\delta}_n' \rangle$ is the facet signature produced for the application, we therefore have $\hat{\sigma}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ \tilde{\sigma}$.

Thus, $(\bigsqcup_{i=1}^{n} \hat{a}_i(\hat{\rho}, \hat{\phi}_n) \sqcup \hat{\sigma})\ \mathcal{R}^{\tilde{\mathcal{A}}}\ (\bigsqcup_{i=1}^{n} \tilde{a}_i(\tilde{\rho}, \tilde{\phi}_n) \sqcup \tilde{\sigma})$. Hence, $App_{\hat{\mathcal{A}}}\ \mathcal{R}^{\tilde{\mathcal{A}}}\ App_{\tilde{\mathcal{A}}}$.

Hence, $\mathcal{R}^{\tilde{\mathcal{A}}}$ holds in general. This concludes the proof. □

Now, we can express the correctness of the global semantic function in the following theorem. Intuitively, the correctness property reflects the fact that if input to a program at facet analysis is an abstraction of the input to the same program at on-line partial evaluation, then the result produced by the facet analysis is also an abstraction of that produced by the partial evaluation.

Theorem 3.7 (Correctness of Global Facet Analysis) *Given a program P in our first-order language. Let $\langle \hat{v}_1, \ldots, \hat{v}_n \rangle$ and $\langle \tilde{\delta}_1, \ldots, \tilde{\delta}_n \rangle$ be initial inputs to P for*

on-line partial evaluation and facet analysis respectively, such that $\hat{v}_i \; \mathcal{R}^{\tilde{\mathcal{A}}} \; \tilde{\delta}_i$, $\forall i \in \{1,\ldots,n\}$. *If* $\hat{\sigma}$ *and* $\tilde{\sigma}$ *are the final caches produced by* $\hat{\mathcal{A}}$ *and* $\tilde{\mathcal{A}}$ *respectively, then* $\hat{\sigma} \; \mathcal{R}^{\tilde{\mathcal{A}}} \; \tilde{\sigma}$.

Proof : Firstly, we notice from the definition of $\tilde{\mathcal{E}}_{Prog}$ that $\langle \mathbf{s}, \tilde{\delta}_1, \ldots, \tilde{\delta}_n \rangle$ is the corresponding facet signature for f_1 in $\tilde{\sigma}$. Therefore, $\langle \mathbf{s}, \tilde{\delta}_1, \ldots, \tilde{\delta}_n \rangle \sqsubseteq \tilde{\sigma}[\![f_1]\!]$. This captures the initial call to the on-line partial evaluation: $\langle \mathbf{s}, \hat{v}_1, \ldots, \hat{v}_n \rangle \in \hat{\sigma}[\![f_1]\!]$. Next $\tilde{h}$ in $\tilde{\mathcal{E}}_{Prog}$ applies $\tilde{\mathcal{A}}$ to each facet signature in the cache, like function $\hat{h}$ in $\hat{\mathcal{E}}_{Prog}$. Since lub operation preserves $\mathcal{R}^{\tilde{\mathcal{A}}}$, $\hat{\sigma} \; \mathcal{R}^{\tilde{\mathcal{A}}} \; \tilde{\sigma}$. □

3.5.3 Deriving the Specialization Semantics

We now describe the derivation of the specialization semantics (for off-line partial evaluation) from its on-line counterpart. This derivation is based on the observation that, prior to on-line partial evaluation, facet analysis has determined the invariants of this process. Indeed, the result of the on-line partial-evaluation computations has been approximated and is available statically. Thus, the aim of this derivation is to transform the on-line partial-evaluation semantics so that it makes use of facet information as much as possible. The uses of facet information are listed below:

1. Predicates testing whether an expression partially evaluates to a constant can safely be replaced by a predicate testing whether this expression is *static* in its binding-time facet.

2. Filter computation for a function call can safely be replaced by an access to the function's facet signature; it contains the call transformation to be performed.

The use of facet information collected for an expression requires that this information be bound to the expression. That is, each expression in a program should be annotated with the information computed by the facet analysis. We achieve this annotation by assigning a unique label to each expression in a program and binding

this label to the corresponding facet information. A cache, noted $\tilde{\psi}$, maps each label of an expression to its facet information. For a label l, we write $(\tilde{\psi}\ l)_v$ to denote the product of abstract facet value corresponding to l. If l is the label of a function call, then $(\tilde{\psi}\ l)_t$ refers to its transformation (*i.e.*, unfolding or suspension).

Note that this annotation strategy only requires a minor change to the core semantics. Namely, the labels of an expression must be passed to the semantic combinator.[8] For example, in specializing a labeled conditional expression $[\![(if\ e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3})^l]\!]$, the combinator $Cond_{\widehat{\mathcal{E}}}$ takes as an additional argument $\langle l, l_1, l_2, l_3\rangle$. Besides passing labels to combinators, we extend the usual pair of environments to include the cache (*i.e.*, $\tilde{\psi} \in \mathbf{AtCache}$).

Figures 3.13 to 3.15 depict the detailed specification of the specialization process. Each interpreted combinator is similar to that of on-line partial evaluation, except in the following cases:

1. For both $Cond_{\widehat{\mathcal{E}}}$ and $Cond_{\widehat{\mathcal{A}}}$, the predicate that determines whether the conditional test evaluates to a constant has been replaced by a predicate that tests the staticity of its binding-time facet value.

2. For primitive call, the predicate testing whether the result of the operation is a constant has been replaced by a predicate testing the staticity of the resulting binding-time facet value.

3. For both $App_{\widehat{\mathcal{E}}}$ and $App_{\widehat{\mathcal{A}}}$, filter computation has been replaced by an access to the static information about the function call: facet value of the arguments and function call transformation.

At this point, it is important to notice that the specialization semantics that we derived indeed describes a specialization process. In fact, as mentioned in [Bondorf *et al.*, 1988,Jones *et al.*, 1989], binding-time analysis was introduced for practical

[8]Note that for simplicity we did not introduce labels in the core semantics presented on page 54. Indeed, labels are only used for the specialization semantics.

reasons. Namely, by taking advantage of binding-time information, the specialization process can be simplified and its efficiency improved. This is a key point for successful self-application [Jones *et al.*, 1989].

Thus, the off-line strategy aims at lifting as many computations as possible from specialization by exploiting static information. In another word, there exists a wide range of specializers for a given language; it depends on how much pre-computation has been performed prior to specializer. In fact, the specialization semantics derived in the previous section may be used as a basis to introduce many optimizations. In the next chapter, we will mention some of these optimizations.

3.6 Discussion

We have presented the semantic specifications and correctness proofs for both on-line and off-line PPE semantics. In doing so, we have addressed and solved a series of open issues in partial evaluation such as relating on-line partial evaluation to standard semantics, showing that facet analysis (and thus binding-time analysis) is an abstraction of the on-line partial-evaluation process, and formally defining the specialization semantics.

Having a well-defined relationship between on-line and off-line PPE enables the transfer of techniques developed in one strategy to another. In particular, we have already seen in this chapter that techniques used in specifying and proving the correctness of the on-line PPE semantics are transferred and used in specifying and proving the correctness of facet analysis. This saves efforts in both the specification process and the proof. Further evidences on transfer of techniques can be found in the implementation (Chapter 4).

Using factorized semantics and logical relations as tools, this work should improve the understanding of partial evaluation. Also, it should provide a basis for implementation. In fact, the specifications presented in this chapter have been implemented using Standard ML. (More discussion about implementation is presented in the next

chapter.)

Partial-evaluation facet plays an important role in the correctness proof of on-line partial evaluation. This is manifested in the correctness proof of the global semantic function $\hat{\mathcal{A}}$. In this respect, we note that the correctness of $\hat{\mathcal{A}}$ only depends on the static information available during partial evaluation, not the residual expressions produced. Since the partial-evaluation facet captures only the processing of static information during partial evaluation, and omits the manipulation of residual expressions, it fits nicely into, and greatly simplifies the proof for $\hat{\mathcal{A}}$. In fact, without partial-evaluation facet, the correctness of $\hat{\mathcal{A}}$ may appear to be dependent of the residual expression produced. Because the generation of some of the residual expressions (the residual calls, in particular) depend on the cache, $\hat{\mathcal{A}}$ and $\hat{\mathcal{E}}$ become circularly dependent of each other ; this makes the correctness proof much more tedious. Using partial-evaluation facet, we can easily specify the semantics and prove the correctness of both conventional on-line and off-line partial evaluation [Consel and Khoo, 1992].

Like other formal work on partial-evaluation specification, we did not address the issue of termination preservation. Instead, we use the notion of filter to specify the function call treatment. The filter described here utilizes the binding-time value of the call arguments to make decision about call transformation. This technique has been used in the conventional partial evaluator Schism [Consel, 1989]. However, it is conceivable that other kind of information can be used in specifying call treatment. In particular, abstract-facet information may be used. Thus, with sign abstract-facet information, we may want to design a filter for a function whose calls will always be unfolded when its arguments are positive.

- Semantic Domains

$$
\begin{array}{rcl@{\qquad}rcl}
l & \in & \textbf{Labels} & & & \\
\hat{\delta} & \in & \widehat{\mathcal{SD}} = \text{as in On-line} & \hat{v} & \in & \mathit{Result}_{\hat{\mathcal{E}}} = \text{as in On-line} \\
\langle \tilde{t}, \tilde{\delta} \rangle & \in & \textbf{Att} = (\textbf{Transf} \times \widetilde{\mathcal{SD}}) & \tilde{\delta} & \in & \widetilde{\mathcal{SD}} = \text{as in Off-line} \\
\tilde{\psi} & \in & \textbf{AtCache} = \textbf{Labels} \rightarrow \textbf{Att} & \hat{\sigma} & \in & \mathit{Result}_{\hat{\mathcal{A}}} = \text{as in On-line} \\
\hat{\rho} & \in & \widehat{\mathit{VarEnv}} = \text{as in On-line} & \hat{\phi} & \in & \widehat{\mathit{FunEnv}} = \text{as in On-line} \\
\widehat{Env} & = & \widehat{\mathit{VarEnv}} \times \widehat{\mathit{FunEnv}} \times \textbf{AtCache} & & & \\
\end{array}
$$

- Valuation Functions

$$
\begin{array}{l}
\hat{\mathcal{E}}_{Prog} : \textbf{Prog} \rightarrow \mathit{Result}^n_{\hat{\mathcal{E}}} \rightarrow \textbf{AtCache} \rightarrow \textbf{Prog}_{\perp} \\
\hat{\mathcal{E}}_{Prog}\ [\![\{f_i(x_1,\ldots,x_n) = e_i\}]\!]\ \langle \hat{v}_1,\ldots,\hat{v}_n \rangle\ \tilde{\psi} = \\
\quad \mathit{MkProg}\ (\hat{h}(\perp[\{\langle \mathbf{s}, \hat{v}_1,\ldots,\hat{v}_n \rangle\}/f_1]))\tilde{\psi}\hat{\phi} \\
\quad \mathit{whererec}\ \hat{h}(\hat{\sigma}) = \hat{\sigma} \sqcup \\
\qquad\qquad \hat{h}(\bigsqcup\{\mathcal{A}_{\widetilde{\mathcal{SA}}}\ [\![e_i]\!](\perp[\hat{v}'_k/x_k], \hat{\phi}, \tilde{\psi}) \mid \langle -, \hat{v}'_1,\ldots,\hat{v}'_n \rangle \in \hat{\sigma}[\![f_i]\!], \\
\qquad\qquad\qquad\qquad [\![f_i]\!] \in \mathit{Dom}(\hat{\sigma})\}) \\
\qquad \hat{\phi} = \perp[\mathit{strict}\ \{\lambda(\hat{v}_1,\ldots,\hat{v}_n)\ .\ \mathcal{E}_{\widetilde{\mathcal{SE}}}\ [\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}, \tilde{\psi})\}/f_i]
\end{array}
$$

- *MkProg* Definition

$$
\begin{array}{l}
\mathit{MkProg}\ \hat{\sigma}\ \tilde{\psi}\ \hat{\phi} = \{\ f_i^{sp}(x_1,\ldots,x_k) = \hat{v}{\downarrow}1 \mid \forall \langle \mathbf{s}, \hat{v}_1,\ldots,\hat{v}_n \rangle \in \hat{\sigma}[\![f_i]\!], \\
\qquad\qquad\qquad\qquad\qquad\qquad \forall [\![f_i]\!] \in \mathit{Dom}(\hat{\sigma})\} \\
\quad \mathit{where}\ f_i^{sp} = \mathit{SpName}([\![f_i]\!], \hat{v}_1,\ldots,\hat{v}_n) \\
\qquad \hat{v}' = \mathcal{E}_{\widetilde{\mathcal{SE}}}\ [\![e_i]\!](\perp[\hat{v}_k/x_k], \hat{\phi}, \tilde{\psi}) \\
\qquad \langle x_1,\ldots,x_k \rangle = \mathit{ResidPars}\ ([\![f_i]\!], \hat{v}_1{\downarrow}1,\ldots,\hat{v}_n{\downarrow}1)
\end{array}
$$

Figure 3.13: Specialization Semantics — Domains and Main functions

- Local Combinator Definitions

$$Const_{\widetilde{\mathcal{SE}}}\ [\![c]\!]\ \langle l\rangle\ =\ \lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \hat{\mathcal{K}}\ [\![c]\!]$$

$$Var_{\widetilde{\mathcal{SE}}}\ [\![x]\!]\ \langle l\rangle\ =\ \lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \hat{\rho}\ [\![x]\!]$$

$$PrimOp_{\widetilde{\mathcal{SE}}}\ [\![p]\!]\ (\hat{k}_1,\ldots,\hat{k}_n)\ \tilde{\delta}\ \langle l,l_1,\ldots,l_n\rangle\ =$$
$$\lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \hat{\mathcal{K}}_{SP}\ [\![p]\!]\ (\hat{k}_1(\hat{\rho},\hat{\phi},\tilde{\psi}),\ldots,\ \hat{k}_n(\hat{\rho},\hat{\phi},\tilde{\psi}))\ (\tilde{\psi}\ l)_v$$

$$Cond_{\widetilde{\mathcal{SE}}}\ (\hat{k}_1,\hat{k}_2,\hat{k}_3)\ \langle l,l_1,l_2,l_3\rangle\ =$$
$$\lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ (\tilde{\psi}\ l_1)^1_v = static\ \rightarrow\ (\mathcal{K}(\hat{v}_1{\downarrow}1)\ \rightarrow \hat{v}_2,\ \hat{v}_3),$$
$$\langle [\![if\ \hat{v}_1{\downarrow}1\ \hat{v}_2{\downarrow}1\ \hat{v}_3{\downarrow}1]\!], \hat{v}_2{\downarrow}2\ \sqcup\ \hat{v}_3{\downarrow}2\rangle$$
$$where\ \ \hat{v}_i\ =\ \hat{k}_i(\hat{\rho},\hat{\phi},\tilde{\psi})\quad \forall i\in\{1,\ 2,\ 3\}$$

$$App_{\widetilde{\mathcal{SE}}}\ [\![f]\!]\ (\hat{k}_1,\ldots,\hat{k}_n)\ \langle l,l_1,\ldots,l_n\rangle\ =$$
$$\lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ (\tilde{\psi}\ l)_t = \mathrm{u}\rightarrow\ \hat{\phi}\ [\![f]\!]\ (\hat{v}_1,\ldots,\hat{v}_n),$$
$$\langle [\![f_{sp}(e''_1,\ldots,e''_k)]\!], \top_{\widehat{\mathcal{SD}}}\rangle$$
$$where\ \hat{v}_i\ =\ \hat{k}_i(\hat{\rho},\hat{\phi},\tilde{\psi})\quad \forall i\in\{1,\ldots,n\}$$
$$f_{sp}\ =\ SpName([\![f]\!],\hat{v}'_1,\ldots,\hat{v}'_n)$$
$$\langle e''_1,\ldots,e''_k\rangle\ =\ ResidArgs\ ([\![f]\!],\ \langle b_1,\ldots,b_n\rangle,\ \langle\hat{v}_1{\downarrow}1,\ldots,\hat{v}_n{\downarrow}1\rangle)$$
$$\langle\hat{v}'_1,\ldots,\hat{v}'_n\rangle\ =\ SpPat\ ([\![f]\!],\langle\hat{v}_1,\ldots,\hat{v}_n\rangle,\ \langle b_1,\ldots,b_n\rangle)$$
$$\langle b_1,\ldots,b_n\rangle\ =\ \langle\tilde{\delta}'^1_1,\ldots,\tilde{\delta}'^1_n\rangle$$
$$\tilde{\delta}'_i = (\tilde{\psi}\ l_i)_v\quad \forall i\in\{1,\ldots,n\}$$

- Primitive Functions

$$\hat{\mathcal{K}}\ :\ \mathbf{Const}\ \rightarrow\ Result_{\widehat{\mathcal{E}}}$$
$$\hat{\mathcal{K}}\ [\![c]\!]\ =\ \text{(as in On−Line Semantics)}$$

$$\hat{\mathcal{K}}_{SP}\ :\ \mathbf{Po}\ \rightarrow\ Result^n_{\widehat{\mathcal{E}}}\ \rightarrow\ \widetilde{\mathcal{SD}}\ \rightarrow\ Result_{\widehat{\mathcal{E}}}$$
$$\hat{\mathcal{K}}_{SP}\ [\![p^c]\!]\ (\langle e'_1,\hat{\delta}_1\rangle,\ldots,\langle e'_n,\hat{\delta}_n\rangle)\ \tilde{\delta}\ =$$
$$(\hat{\delta}\ =\ \bot_{\widehat{\mathcal{D}}})\rightarrow\langle\bot_{Exp},\bot_{\widehat{\mathcal{SD}}}\rangle,$$
$$(\tilde{\delta}^1 = static)\rightarrow\ \langle\hat{\delta}^1,\langle\hat{\alpha}_{\widehat{\mathcal{D}}^1}(d),\ldots,\ \hat{\alpha}_{\widehat{\mathcal{D}}^m}(d)\rangle\rangle,\ \langle e',\hat{\delta}\rangle$$
$$where\ p^c\ :\ \mathbf{D}^n\ \rightarrow\ \mathbf{D}$$
$$\hat{\delta} = \hat{\omega}_{p^c}(\hat{\delta}_1,\ldots,\hat{\delta}_n)$$
$$e' = [\![p^c(e'_1,\ldots,e'_n)]\!]$$
$$d = \mathcal{K}(\hat{\delta}^1)$$

$$\hat{\mathcal{K}}_{SP}\ [\![p^o]\!]\ (\langle e'_1,\hat{\delta}_1\rangle,\ldots,\langle e'_n,\hat{\delta}_n\rangle)\ \tilde{\delta}\ =$$
$$(\hat{d}\ =\ \bot_{\widehat{Values}})\rightarrow\langle\bot_{Exp},\bot_{\widehat{\mathcal{SD}}}\rangle,$$
$$(\tilde{\delta}^1 = static)\rightarrow\ \langle\hat{d},\langle\hat{\alpha}_{\widehat{\mathcal{D}}'^1}(d),\ldots,\hat{\alpha}_{\widehat{\mathcal{D}}'^m}(d)\rangle\rangle,\ \langle e',\langle\top_{\widehat{\mathcal{D}}'^1},\ldots,\ \top_{\widehat{\mathcal{D}}'^m}\rangle\rangle$$
$$where\ p^o\ :\ \mathbf{D}^n\ \rightarrow\ \mathbf{D}'$$
$$\hat{d} = \hat{\omega}_{p^o}(\hat{\delta}_1,\ldots,\hat{\delta}_n)$$
$$e' = [\![p^o(e'_1,\ldots,e'_n)]\!]$$
$$d = \mathcal{K}(\hat{d})$$

Figure 3.14: Specialization Semantics — Local semantic function

- Global Combinator Definitions

$$Const_{\widetilde{SA}}\ [\![c]\!]\ \langle l\rangle\ =\ \lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \lambda l\ .\ \bot_{Att}$$

$$Var_{\widetilde{SA}}\ [\![x]\!]\ \langle l\rangle\ =\ \lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \lambda l\ .\ \bot_{Att}$$

$$PrimOp_{\widetilde{SA}}\ [\![p]\!]\ (\hat{a}_1,\ldots,\hat{a}_n)\ \langle l,l_1,\ldots,l_n\rangle\ =\ \lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ \bigsqcup_{i=1}^{n}\hat{a}_i(\hat{\rho},\hat{\phi},\tilde{\psi})$$

$$Cond_{\widetilde{SA}}\ (\hat{a}_1,\hat{a}_2,\hat{a}_3)\ \hat{k}_1\ \langle l,l_1,l_2,l_3\rangle\ =$$
$$\lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ (\hat{a}_1(\hat{\rho},\hat{\phi},\tilde{\psi}))\sqcup$$
$$((\tilde{\psi}\ l)_v^1 = static)\ \rightarrow\ \mathcal{K}(\hat{\delta}_1^1)\rightarrow \hat{a}_2(\hat{\rho},\hat{\phi},\tilde{\psi}),\hat{a}_3(\hat{\rho},\hat{\phi},\tilde{\psi}),$$
$$\hat{a}_2(\hat{\rho},\hat{\phi},\tilde{\psi})\sqcup\hat{a}_3(\hat{\rho},\hat{\phi},\tilde{\psi}))$$

$$App_{\widetilde{SA}}\ [\![f]\!]\ (\hat{a}_1,\ldots,\ \hat{a}_n)\ (\hat{k}_1,\ldots,\ \hat{k}_n)\ \langle l,l_1,\ldots,l_n\rangle\ =$$
$$\lambda(\hat{\rho},\hat{\phi},\tilde{\psi})\ .\ (\bigsqcup_{i=1}^{n}\hat{a}_i(\hat{\rho},\hat{\phi},\tilde{\psi}))\sqcup\hat{\sigma}$$
$$where\ \hat{v}_i\ =\ \hat{k}_i(\hat{\rho},\hat{\phi},\tilde{\psi})\quad\forall i\in\{1,\ldots,n\}$$
$$\hat{\sigma}\ =\ (\tilde{\psi}\ l)_t = \mathbf{u}\rightarrow\ \bot[\{\langle \mathbf{u},\hat{v}_1,\ldots,\hat{v}_n\rangle\}/f],\ \bot[\{\langle \mathbf{s},\hat{v}'_1,\ldots,\hat{v}'_n\rangle\}/f]$$
$$\langle\hat{v}'_1,\ldots,\hat{v}'_n\rangle\ =\ SpPat\ ([\![f]\!],\langle\hat{v}_1,\ldots,\hat{v}_n\rangle,\ \langle b_1,\ldots,b_n\rangle)$$
$$\langle b_1,\ldots,b_n\rangle\ =\ \langle\tilde{\delta}'^1_1,\ldots,\tilde{\delta}'^1_n\rangle$$
$$\tilde{\delta}_i\ =\ (\tilde{\psi}\ l_i)_v\quad\forall i\in\{1,\ldots,n\}$$

Figure 3.15: Specialization Semantics — Global semantic function

Chapter 4

Implementation

Techniques for implementing partial evaluation abound. Throughout the years, various optimization techniques have been applied to the partial-evaluation process. While these techniques improve partial-evaluation efficiency, they also make the partial-evaluation process less comprehensible. Due to the lack of theoretical foundation, many implementations have been based on the intuitions of the implementors, instead of a provably-correct semantic model. This naturally raises doubt about the correctness of such large software systems. Furthermore, it hinders the introduction of new optimization techniques, since it becomes difficult to reason about the effect of these optimizations on the correctness of the system.

The semantic specification of PPE described in the previous chapters provides the desired foundation for implementation. In this chapter, we first show the implementations of both on-line and off-line PPE based on our semantic model. Secondly, we describe some optimization techniques. Lastly, to demonstrate the effectiveness of PPE, we present some applications and assess their performance.

4.1 Current Implementation

We have implemented both on-line and off-line PPE for a first-order applicative language with data structures. This language is very much in the style of other first-order functional languages. It is monomorphically typed. A program consists of a series of data-type declarations followed by function definitions. The first function defined is the main function of the program.

Each function consists of four parts: a name, parameters, an optional filter specification and an expression. Filter specification controls the treatment of calls to the function. It is described in Section 3.4.2. When it is left out, every call to the function will be systematically unfolded.

Two kinds of algebraic data types are available: union and product. A union type is declared using the `DataType` construct, and a product type using the `Type` construct. Three operators define manipulation of data structures: `DCons` constructs a data structure, `Sel` selects a component from a structure, and `CaseType` acts like a conditional construct: it allows selection of a branch using the data-constructor name of the test. Following is an example of a typical program written in this language:

```
datatype Stack = Empty | Entry of (Int,Stack)
Fun SumStack (x)
    caseType x
     (Empty => 0
      else  =>
       (let ( [i (Sel x 1)] )
         (addInt i (Sumstack (Sel x 2)))))
```

Although programs written in this language are assumed to be well-typed, we have not implemented the type inference system. To aid the partial evaluator in determining the type of each expression, type information is attached to the expression. Thus, the above program is implemented as follows:

```
datatype Stack = Empty | Entry of (Int,Stack)
Fun SumStack (x [Stack])
```

```
caseType x
 (Empty => 0
  else  =>
   (let ( [i (Sel [int] x 1)] )
     (prim [int] addInt i
           (call [int] Sumstack (Sel [Stack] x 2)))))
```

Certainly, with a type inference system installed, we can safely discard the type annotation. Therefore, for clarity, all programs shown in this thesis are without type information.

Recall from Definition 2.2 that the constant Domain $\widehat{\mathbf{Values}}$ contains the textual representation of all basic values provided by our first-order language. It is defined by data type `Value` as follows (note that the Standard ML datatype notation [Milner *et al.*, 1990] is used in declaring data types):

```
datatype Value = Num of int | Bool of bool | Str of string
               | Constr of string * string * (Value list)
```

Data constructor `Constr` captures other kind of basic values that are of algebraic nature, such as the vector domain. It also captures the representation of static properties. For example, the size property of a vector having size 3, viewed as a static property, can be represented as `Constr("Vect", "Size", [Num(3)])`.

Recall from Chapter 3 that the local semantic function of on-line PPE operates partly on product of facets ($\widehat{\mathcal{SD}}$). In this implementation, it also operates on data structures. Thus, the actual domain used is defined as data type `DSValue`:

```
datatype DSValue =
  PFCons of string * (Value list) |       (* product-of-facet value *)
  Prod of string * (DSValue list) |       (* product type *)
  Sum of string * string * (DSValue list) (* union type *)
```

The first field (of type `string`) associated with each data constructor of `DSValue` contains the value's type. The second field associated with constructor `Sum` contains the data-constructor name of the union type.

Data type DSValue constitutes the second component of the result domain for the local semantic function. The first component is the domain for residual expressions, which is also the syntactic domain processed by both the local and global semantic functions. It is defined by data type Exp:

```
datatype Exp = Var of string * string | Const of Value * string
             | Prim of string * (Exp list) * string
             | Cond of Exp * Exp * Exp * string
             | Call of string * (Exp list) * string
             | Let of (string list) * (Exp list) * Exp * string
             | DCons of string * (Exp list) * string
             | Sel of Exp * Exp * string
```

This set of possible expressions extends the one described in Figure 3.2 (for a first-order language) to include a Let construct and primitives for data structure manipulation. These primitives are: DCons constructs a data structure and Sel selects a field from a structure. Incidentally, the construct CaseType that appears in the subject language can be translated into other more elementary expressions (*i.e.*, conditional and structure-field-selection expressions).

Note that every expression construction ends with a field of type string. This field contains the type information about the expression. Retaining type information is important because it helps choose the corresponding primitive operations from a list of products of facets.

4.1.1 On-Line PPE

Our parameterized partial evaluator is implemented in Standard ML [Milner *et al.*, 1990]. The semantic specification given in Chapter 3 can be readily translated into a program. Several modifications are made to the specification:

1. The meta-language describing the semantics is a lazy language, whereas the implementation language is strict. The problem is resolved by converting the semantics into continuation-passing style [Steele, 1978,Danvy and Filinski, 1991].

This rids the partial evaluator of evaluation order. (Such conversion algorithms are already available, for example, in [Danvy and Filinski, 1991].)

2. Partial evaluation of the Let construct is included. This involves partially evaluating those expressions associated with local variables, updating the local environment, and partially evaluating the Let body. When some of the local variables defined are not assigned to constants, the Let construct remains residual in order to capture the binding of these dynamic variables.

3. Partial evaluation of a data structure construction simply returns the constructed structure. Partial evaluation of a structure-field selection returns the selected field when it is present. Otherwise, the selection operation is frozen until run-time.

Implementing Facets

The implementation is modular; that is, the partial evaluator consists of several modules, each of which can be tested independently before being put together to form the system. The Standard ML module construct directly supports modular design. Specifically, the user of our PPE implementation defines facets using Standard ML modules. These facets are then tested before being used by the parameterized partial evaluator. A facet has the following signature:

```
signature FACET =
   sig
    val bot : Value                                          (* bottom *)
    val top : Value                                             (* top *)
    val Operators : string list                          (* operator list *)
    val abs : Value -> Value                        (* abstraction function *)
    val lub : Value * Value -> Value                        (* lub operator *)
    val glb : Value * Value -> Value                        (* glb operator *)
    val opType : string -> OpClass             (* operator classification *)
    val cop : string -> (Value list) -> Value (* closed operators *)
    val oop : string -> (Value list) -> Value   (* open operators *)
   end
```

Notice that, for a facet $\widehat{\mathbf{D}}$, both $\bot_{\widehat{D}}$ and $\top_{\widehat{D}}$ are specified explicitly in the definition. This is necessary, since the former will be used in defining the product of facets, and the latter will be used by the partial evaluator. Because the facet domain is a complete lattice, we need to define both the lub and glb operators. The operator classification indicates whether an operator is closed or open. Following is the Standard ML definition of a facet module called `VecSIZE` that defines the vector-size property:

```
structure VecSIZE : FACET =
 struct
  val bot = Constr("Vect","Bot",[])
  val top = Constr("Vect","Top",[])
  val Operators = ["MKVEC","UPDVEC","NUMVEC","REFVEC","EQVEC"]
  fun abs (Constr("Vect","VEC",_)) =
                Constr("Vect","Size",[Array.oop "NUMVEC" [a]])
    | abs _ = Constr("Vect","Top",[])
  fun lub (Constr("Vect","Bot",[]),y) = y
    | lub (x,Constr("Vect","Bot",[])) = x
    | lub (x as Constr(_,"Size",_),y as Constr(_,"Size",_)) =
        if (x=y) then x else top
    | lub (_,_) = top
  fun glb (Constr("Vect","Top",[]),y) = y
    | glb (x,Constr("Vect","Top",[])) = x
    | glb (x as Constr(_,"Size",_),y as Constr(_,"Size",_)) =
        if (x=y) then x else bot
    | glb (_,_) = bot
  fun opType "MKVEC"  = Closed
    | opType "UPDVEC" = Closed
    | opType "NUMVEC" = Open
    | opType "REFVEC" = Open
    | opType "EQVEC"  = Open
  fun cop "MKVEC" [Constr(_,"Bottom",[])] = Constr("Vect","Bot",[])
    | cop "MKVEC" [Constr(_,"Top",[])] = Constr("Vect","Top",[])
    | cop "MKVEC" [n as (Num(_))] = Constr("Vect","Size",[n])
    | cop "UPDVEC" [Constr(_,"Bot",_),_,_] = bot
    | cop "UPDVEC" [_,Constr(_,"Bottom",_),_] = bot
    | cop "UPDVEC" [_,_,Constr(_,"Bottom",_)] = bot
    | cop "UPDVEC" [Constr(_,"Top",_),_,_] = top
```

```
    | cop "UPDVEC" [a as Constr(_,"Size",_),_,_] = a
  fun oop "NUMVEC" [Constr(_,"Bot",_)] = Constr("INT","Bottom",[])
    | oop "NUMVEC" [Constr(_,"Top",[])] = Constr("INT","Top",[])
    | oop "NUMVEC" [Constr(_,"Size",[n])] = n
    | oop "REFVEC" [Constr(_,"Bot",_),_] = Constr("INT","Bottom",[])
    | oop "REFVEC" [_,Constr(_,"Bottom",_)] =
                          Constr("INT","Bottom",[])
    | oop "REFVEC" [Constr(_,"Top",_),_] = Constr("INT","Top",[])
    | oop "REFVEC" [_,Constr(_,"Top",_)] = Constr("INT","Top",[])
    | oop "REFVEC" [x,y] = Constr("INT","Top",[])
    | oop "EQVEC" [Constr(_,"Bot",_),_] = Constr("BOOL","Bottom",[])
    | oop "EQVEC" [_,Constr(_,"Bot",_)] = Constr("BOOL","Bottom",[])
    | oop "EQVEC" [Constr(_,"Top",_),_] = Constr("BOOL","Top",[])
    | oop "EQVEC" [_,Constr(_,"Top",_)] = Constr("BOOL","Top",[])
    | oop "EQVEC" [Constr(_,"Size",[Num(n)]),
                   Constr(_,"Size",[Num(m)])] =
       if n <> m then Bool(false) else Constr("BOOL","Top",[])
    | oop "EQVEC" [x,y] = Constr("BOOL","Top",[])
  end
```

Although a good modular tool, a Standard ML module is not a first-class object. Therefore, facets defined as modules cannot be integrated to form a product of facets. This problem is circumvented by converting a facet module into a Standard ML data structure of type FacetType defined as follows:

```
datatype FacetType =
   Facet of string                                    (* Facet name *)
         * Value * Value                            (* bottom and top *)
         * string list                            (* list of operators *)
         * (Value -> Value)                   (* abstraction function *)
         * (Value * Value -> Value)                 (* lub operation *)
         * (Value * Value -> Value)                 (* glb operation *)
         * (string -> OpClass)             (* operator classification *)
         * (string -> (Value list) -> Value) (* Closed operators *)
         * (string -> (Value list) -> Value)   (* Open Operators *)
```

The data structure capturing the vector-size facet, called `size`, can therefore be obtained by the following binding:

```
val size = let open VecSIZE in
            Facet("VecSIZE",bot,top,Operators,
                          abs,lub,glb,opType,cop,oop)
           end ;
```

Implementing Product of Facets

Facets are grouped together to form a product of facets. A product of facets is implemented as a data structure of type `ProdOfFacets`:

```
datatype ProdOfFacets =
  PFacet of string                                (* ProdOfFacet name *)
         * int                                    (* number of facets *)
         * DSValue * DSValue                        (* bottom and top *)
         * string list                            (* list of operators *)
         * (Value -> DSValue)                  (* abstraction function *)
         * (DSValue * DSValue -> DSValue          (* lub operation *)
         * (DSValue * DSValue -> DSValue          (* glb operation *)
         * (string -> OpClass)           (* operator classification *)
         * (string -> (DSValue list) -> DSValue)    (* closed ops *)
         * (string -> (DSValue list) -> string -> DSValue)
                                                      (* open ops *)
```

Notice that open operators take one more argument (of type `string`) than closed operators. This argument contains the type of the result of the open operation.

Applying a list of data structures of `FacetType` (*i.e.*, facets) to the function `mkPFacet` creates a data structure representing a product of facets. Function `mkPFacet` defines each component of the product of facets from the corresponding components of the constituents. Furthermore, it defines the product-of-facet operations.

```
fun mkPFacet (name, flst) = (* PE facet must be the head of flst *)
 let val pfnum = length(flst)
     fun iterate [] lls = lls
       | iterate ((Facet(_,b,t,Op,a,l,g,ot,c,pn))::[])
                 (bs,ts,Ops,abss,ls,gs,ots,cs,pns) =
         (rev(b::bs),rev(t::ts),Op::Ops,rev(a::abss),rev(l::ls),
            rev(g::gs),ot::ots,rev(c::cs),rev(pn::pns))
       | iterate ((Facet(_,b,t,_,a,l,g,_,c,pn))::fs)
```

```
            (bs,ts,Ops,abss,ls,gs,ots,cs,pns) =
    iterate fs (b::bs,t::ts,Ops,a::abss,
                l::ls,g::gs,ots,c::cs,pn::pns)
val (bs,ts,Ops,abss,ls,gs,ots,cs,pns) =
      iterate flst ([],[],[],[],[],[],[],[],[])
val pfbot = PFCons(name,bs)
val pftop = PFCons(name,ts)
fun mkDh x = let val (PFCons(_,bots)) = pfbot
                 val bs = map (fn (x1,x2) => x1 = x2)
                              (zip x bots)
             in if (foldl (fn x => fn y => x orelse y)
                          false bs)
                then pfbot
                else (PFCons(name,x))
             end
fun abs x = PFCons(name,(map (fn a => a(x)) abss))
fun lub (PFCons(_,x),PFCons(_,y)) =
     mkDh (map (fn (f,pair) => f(pair)) (zip ls (zip x y)))
fun glb (PFCons(_,x),PFCons(_,y)) =
     mkDh (map (fn (f,pair) => f(pair)) (zip gs (zip x y)))
fun marknonCarrier [] = []
  | marknonCarrier ((pfv as PFCons(s,x))::pfvs) =
    if (s = name) then pfv::(marknonCarrier pfvs)
     else PFCons("OPEN",repeat (hd(x)) pfnum)::
          (marknonCarrier pfvs)
fun cop opname [] = mkDh (map (fn f => (f opname [])) cs)
  | cop opname args =
     let val vs = map (fn (PFCons(_,x)) => x)
                      (marknonCarrier args)
         val args' = zipstar vs
         val (r as PFCons(_,pe::_)) =
              mkDh (map (fn (f,ars) => (f opname ars))
                        (zip cs args'))
      in case pe of Constr(_,"Bottom",_) => r
                  | Constr(_,"Top",_) => r
                  | _ => abs(pe)
     end
fun omega0 [] v ty = v
  | omega0 (v::vs) cv ty =
    case v of Constr(_,"Bottom",_) => v
```

```
                | Constr(_,"Top",_) => omega0 vs cv ty
                | _ => omega0 vs v ty
    fun oop opname [] ty =
         let val opres = map (fn f => (f opname [])) pns
          in PFCons("OPEN",[omega0 opres (Constr(ty,"Top",[])) ty])
         end
      | oop opname args ty =
         let val vs = map (fn (PFCons(_,x)) => x)
                          (marknonCarrier args)
             val args' = zipstar vs
             val opres = map (fn (f,ars) => (f opname ars))
                             (zip pns args')
          in PFCons("OPEN",[omega0 opres (Constr(ty,"Top",[])) ty])
         end
 in PFacet(name,pfnum,pfbot,pftop,hd(0ps),
           abs,lub,glb,hd(ots),cop,oop)
end ;
```

We have already seen how the data structure `size` for the vector-size facet is obtained. Similarly, we can create a data structure (name it `vectPE`) for the partial-evaluation facet of vector. Using function `mkPFacet`, we can now produce the data structure representing a product of facets for vector; it contains both the partial-evaluation facet and the vector-size facet:

```
val vectPF = mkPFacet (''VECTOR'',[vectPE,size]) ;
```

4.1.2 Off-Line PPE

Implementation of off-line PPE is analogous to the on-line case. The semantic specification given in Chapter 3 is used once again to guide the implementation. In particular, facet analysis has been extended to handle binding-time information about data structures; this information is called partially-static structures [Mogensen, 1989].

Techniques for handling partially-static structures are available in the literature. In this implementation, we adopt the technique described in [Consel, 1990a]. This technique attaches a label (called a *cons-point*) to each data constructor occurred in

a program. It manipulates these cons-points when analyzing data structures. Cons-points are represented by integers; they are assigned uniquely to each data constructor when the program is parsed. Since the number of cons-points used is determined by the number of data constructors appeared textually in the (finite) program, it is finite.

Facet analysis now operates on the set of cons-points in addition to the domain $\widetilde{\mathcal{SD}}$. This enhanced domain is implemented as a data type `DAValue`:

```
datatype DAValue =
  FACons of string * (Value list) (* prod-of-abs-facet value *)
  | Conspt of string * (int list) (* data structure *)
```

The first field (of type `string`) associated with each data constructor contains its type. Notice from the representation of the product-of-abstract-facet value that all static properties are represented by the data type `Value`, just like those in the on-line counterpart. Data constructor `Conspt` works as follows: during facet analysis, when this constructor is assigned to a variable (or an expression) in the program, its integer-list field represents the set of possible cons-points that may be assigned to this variable (or expression); that is, the set of possible data constructors that may be assigned to this variable at specialization time.

Notice that facet analysis depends on type information, whereas binding-time analysis does not.[1] This means that there is no one single value that represents *any* static value occurring in the program. Consequently, input description of facet analysis cannot simply be taken from the binding-time domain $\widetilde{\mathbf{Values}}$, as is the case with binding-time analysis.

As is the case with facets, abstract facets are defined using Standard ML modules. Once tested, an abstract facet is converted to a data structure of type `AbsFacetType`. Analogously, a product of abstract facets is implemented as a data structure of type `ProdofAbsFacets`. These two types are defined as follows:

[1] Regardless of whether a program is typed, binding-time analysis only deals with binding-time values. In facet analysis, however, it is necessary to retain type information, since values of different types may have (and usually do have) different product-of-abstract-facet values.

```
datatype AbsFacetType =
   AFacet of string                          (* Abstract Facet name *)
          * Value * Value                        (* bottom and top *)
          * string list                        (* list of operators *)
          * (Value -> Value)                 (* abstraction function *)
          * (Value * Value -> Value)               (* lub operation *)
          * (Value * Value -> Value)               (* glb operation *)
          * (string -> OpClass)          (* operator classification *)
          * (string -> (Value list) -> Value)      (* Closed ops *)
          * (string -> (Value list) -> Value)        (* Open Ops *)

datatype ProdOfAbsFacets =
   PAFacet of string                          (* ProdOfAbsFacet name *)
           * int                                 (* number of facets *)
           * DAValue * DAValue                     (* bottom and top *)
           * string list                         (* list of operators *)
           * (DSValue -> DAValue)              (* abstraction function *)
           * (DAValue * DAValue -> DAValue)      (* lub operation *)
           * (DAValue * DAValue -> DAValue)      (* glb operation *)
           * (string -> OpClass)       (* operator classification *)
           * (string -> (DAValue list) ->  DAValue)
                                                    (* closed ops *)
           * (string -> (DAValue list) -> string -> DAValue)
                                                      (* open ops *)
```

Applying a list of abstract facets to the function **mkPAFacet** creates a product of abstract facets. This function is similar to the function **mkPFacet** used in producing product of facets; its definition is omitted here.

Using the result of facet analysis, we annotate the expressions in a program with abstract-facet information. The annotated expression is represented by data type **AnnExp**. This type is similar to **Exp**, except that each data construct is associated with an additional field, of type **DAValue**, to capture the abstract-facet information. The specializer now takes the annotated program as input and returns a residual program without annotation. The specializer is as specified in Chapter 3.

4.1.3 Optimizations

While the semantic specifications capture the essence of PPE and guide their implementations, they do not address the efficiency aspect of partial evaluation. In fact, a direct implementation of the specification would result in a partial evaluator that is unreasonably slow. This is mainly due to repetitive computations in the course of partial evaluation. For instance, two identical function calls in a program are partially evaluated independently even though the result of partially evaluating the second call can be readily obtained from the result of partially evaluating the first one. In order to eliminate repetitive computations, and to obtain an efficient partial evaluator in general, various optimizations are needed.

Since PPE is a natural extension of conventional partial evaluation, optimization techniques employed by the latter can also be applied to the former with minor modifications. Moreover, improvements obtained from these techniques can also be realized in PPE. Therefore, some, but not all, optimization techniques have been included in the current implementation.

An important optimization that eliminates many repetitive computations is to make the cache containing partial-evaluation signatures available during partial evaluation of an expression. When suspending a function application during partial evaluation, the partial evaluator first computes the function arguments. It then looks in the cache for any partial-evaluation signature that matches the current set of arguments. If a match occurs, the result of suspending the application is retrieved directly from the cache; otherwise, the function of the application is specialized, and the cache is updated to record this new partial-evaluation signature. This optimization eliminates repetitive partial evaluations of those function applications that are to be suspended. Almost all existing partial evaluators employ a variant of such optimization.

Another optimization included is the technique of *depth-first specialization with pending list*: when a function application is encountered, if its argument-set fails to match any partial-evaluation signature in the cache, the function body is immediately

partially evaluated (thus the term depth-first). Furthermore, a pending list is kept during partial evaluation; this list contains those calls that are currently activated for specialization, but not yet completed. To determine a new recursive suspended call, the partial evaluator only needs to look on the pending list for a suspended call that matches the arguments of the current application. This optimization technique is commonly used in conventional partial evaluation — be it on-line or off-line. A detailed discussion can be found in [Weise and Ruf, 1990].

Two other optimization techniques included in current implementation are the following:

1. *Re-use specialization.* This optimization aims at reducing the number of specialized functions in the residual program. It determines sets of specialized functions with different specialization patterns but identical specialized body. It then replaces functions in each set by one representative function; that is, re-using the representative function.

   This technique was first presented in [Ruf and Weise, 1991] for the partial evaluator Fuse. It determines the opportunity for re-using an existing specialized function *before* specializing a function. As such, this technique reduces partial-evaluation time. On the other hand, since it speculates that re-use is possible before specializing a function, not all specialized functions within a set can be detected and replaced by the representative.

   A similar optimization is also found in the field of compilation, under the name of *procedure cloning* [Cooper *et al.*, 1992]. The goal of procedure cloning is to produce specialized versions of procedures, each dealing with different input data. Here, different versions of a procedure may have different input data but the same procedure body. In [Cooper *et al.*, 1992], the decision to re-use a procedure is performed *after* the specialized versions are created. This does not save compilation time, but it can identify more re-use opportunity than the approach of [Ruf and Weise, 1991].

Both the techniques described above can be implemented in our partial evaluator. Since the second technique requires less modification to the existing system, and detects more re-use opportunity than the first one, we implemented the second approach in the current system.

2. *Eliminating useless facet computations.* This optimization applies to off-line PPE. Even though only useful facets are introduced during partial evaluation, not all these facets are needed throughout the whole program. For example, if it is determined at analysis time that a particular facet computation will produce a constant, then it is not necessary to perform computation on other facets in the same product at specialization time. On the other hand, if it is determined at analysis time that every facet computation of a primitive operation will return the top element in its respective facet domain, then it is not necessary to perform these facet computations at specialization time. On a larger scale, if it is detected that a particular function will not utilize any facet information, and will always return totally dynamic result, then facet computations will not be needed when specializing this function.

   Facet analysis allows us to determine statically which facet computations will produce constants at specialization time. We can exploit this information to eliminate, prior to specialization, the facet computations that do not produce static values.

   In the current implementation, we identify, for each static expression in a program, those abstract facets which actually produce static values. (There can be more than one such abstract facets.) We thus determine a minimal set of abstract facets needed to produce all the static values in the program. In doing so, we eliminate the time spent on computing irrelevant facet information at specialization time.

There are other existing optimization techniques that can also be included in the implementation. For example, at the off-line partial evaluation level, *specialization-*

action analysis described in [Consel, 1989,Consel and Danvy, 1990] can be used to infer statically the actions to be performed by the specializer. The basic specialization actions include standard evaluation of an expression and performing no computation on an expression. These actions can be determined using the abstract-facet value of an expression.

Finally, in comparison with the existing partial evaluators which have static properties built in, the framework of PPE can be considered a kind of optimization. This is because by including only those facets pertaining to an application of partial evaluation, we have already eliminated a lot of static-property computations irrelevant to the given program.

4.2 Some Applications

Using static information about program input, PPE enhances conventional partial evaluation by producing more dedicated residual programs. In Chapter 2, we have seen how to use vector-size information during the parameterized partial evaluation of an Inner-product program to produce a linear residual code. In this section, we provide two more applications to further demonstrate the effectiveness of this new form of partial evaluation. Specifically, we apply parameterized partial evaluation to specialize a pattern matcher and to specialize an interpreter. Both applications result in qualitative improvement in the residual programs produced. Because we have chosen these applications such that both on-line and off-line PPE's produce identical residual programs, we only discuss the on-line case here.

4.2.1 Improving Pattern Matching

In this section, we apply PPE to specialize a pattern matcher with respect to a subject string and a partially-known pattern string. The experiment is based on a modified version of the pattern matcher for strings in [Consel and Danvy, 1989]. Figures 4.1

```
Fun kmp (p d)
  Filter ((if (or (isDyn d) (isDyn p)) Specialize Unfold) (Par Par))
  caseType p (Nil => true
              x  => (start p d))
Fun start (p d) Void
  caseType d (Nil => false
              x => (restart p d))
Fun restart (p d) Void
  (if (eqInt (Sel p 1) (Sel d 1))
      casetype (Sel p 2)
      (Nil => true
       x => caseType (Sel d 2)
            (Nil => false
             x => (loop (Sel p 2) (Sel d 2) p)))
      (start p (Sel d 2)))
```

Figure 4.1: Pattern Matcher — Part 1

and 4.2 display this modified version. It deals with integers instead of characters.

The pattern matcher has the following known effect: partially evaluating the pattern matcher with respect to a pattern string produces a residual program that has the effect of the KMP algorithm [Consel and Danvy, 1989]. This result can be realized in the context of PPE without facet information (except, of course, the partial-evaluation facet).

Suppose that we know some information about the subject string. In particular, suppose that the pattern string is `[2,1]` and the subject string is `[3,x,1,2,y,3,x,y,6, y,1,x,x,1]`, where variable `x` is known to be either `1` or `3`, and variable `y` is known to be either `2` or `4`. Then, it is possible to produce a more specialized residual program by taking into account the possible values of `x` and `y`.

Firstly, consider the case of conventional partial evaluation, which cannot capture the information about the possible values of `x` and `y`. The following residual program is produced by conventional partial evaluation:

```
Fun MAIN (mx5 mx7)
```

```
Fun loop (p d pp)
  Filter ((if (isDyn d) Specialize Unfold) (Par Par Par))
  (if (eqInt (Sel p 1) (Sel d 1))
      caseType (Sel p 2)
      (Nil => true
       x => caseType (Sel d 2)
            (Nil => false
             x => (loop (Sel p 2) (Sel d 2) p)))
      (let ([np (skmp pp (Sel pp 2)
                         (addInt (length (Sel pp 2))
                                 (negInt (length p))))])
        (if (eqlst np pp)
            (if (eqInt (Sel pp 1) (Sel p 1))
                (start pp (Sel d 2))
                (restart pp d))
            (loop np d pp))))
Fun skmp (p d n) Void
  (sloop p d n p d n)
Fun sloop (p d n pp dd nn)
  Void
  (if (eqInt n 0)
      p
      (if (eqInt (Sel p 1) (Sel d 1))
          (sloop (Sel p 2) (Sel d 2) (addInt n (~1)) pp dd nn)
          (skmp pp (Sel dd 2) (addInt nn (~1)))))
Fun length (l) Void
  CaseType l (Nil => 0
              x => (addInt 1 (length (Sel l 2))))
Fun eqlst (a b) Void
  casetype a
  (Nil => (CaseType b (Nil => true
                       x => false))
   x => caseType b
        (Nil => false
         y => (andBool (eqInt (Sel a 1) (Sel b 1))
                       (eqlst (Sel a 2) (Sel b 2)))))
```

Figure 4.2: Pattern Matcher — Part 2

```
    (KMP1 mx5 mx7 mx5 mx7 mx7 mx5 mx5)
Fun KMP1 (x1 x2 x3 x4 x5 x6 x7)
    (If (EQINT 2 x1) True
        (If (EQINT 1 x2) True
            (If (EQINT 2 x3)
                (If (EQINT 1 x4) True
                    (If (EQINT 2 x5) True
                        (If (EQINT 2 x6)
                            (If (EQINT 1 x7) True
                                (If (EQINT 2 x7) True False))
                            (If (EQINT 2 x7) True False))))
                (If (EQINT 2 x5) True
                    (If (EQINT 2 x6)
                        (If (EQINT 1 x7) True
                            (If (EQINT 2 x7) True False))
                        (If (EQINT 2 x7) True False))))))
```

In PPE, we take into consideration the set of possible values assigned to variables x and y. This is achieved by defining a *possible-value facet.* This facet captures the set of possible values assigned to a variable; it also defines the primitive operations on value-sets, rather than usual values. The residual program produced by the parameterized partial evaluator is:

```
Fun MAIN (mx1 mx3)
       (KMP1 mx1 mx3 mx1 mx3 mx3 mx1 mx1)
Fun KMP1 (x1 x2 x3 x4 x5 x6 x7)
    (If (EQINT 2 x5) True False)
```

This result shows that, by taking into account static information about the dynamic values, the partial evaluator, with the help of facet computation, can reduce many conditional tests in the program to constants. Consequently, many conditional expressions can be reduced to one of the branches, and the residual program produced become more dedicated than the one produced by conventional partial evaluation.

A similar result can be produced by the Redfun system [Haraldsson, 1977], which has the possible-value operation built in. However, we note that operations over sets of values can be considerably more time consuming than operations over values. In

other applications where information about set of values is not required, Redfun will nevertheless spend time computing these operations.

4.2.2 Compiling Untyped Programs

It is well-known that partially evaluating an interpreter with respect to a program produces a residual program in which some interpretive overhead is removed. In the context of untyped language, an interpreter for such language usually contains code that performs type checking on the program it is interpreting. However, a conventional partial evaluator that specializes such an interpreter with respect to a program only removes the type-checking code from static expressions. This is because type information is a static property about a value; it can only be determined from known values, not from dynamic values. Our goal in this section is to perform as much type-checking-code removal as possible by introducing a *type facet* into PPE.

First, we set up the right environment for partial evaluation. Notice that two programs are read in before partial evaluation: the interpreter and the program to be interpreted. They are read in using the same parser. In an untyped program, basic values do not possess explicit type information. Since our partial evaluator specializes typed programs, in order to have it specialized untyped programs, we introduce into the partial evaluator a built-in data structure called `Univ` (for universal type) which captures all basic values. We also modify the parser so that basic values in the parsed program are turned into values of universal type. The result is a partial evaluator for untyped programs.

Figures 4.3 and 4.4 show the data-type declarations and main functions used by the interpreter. Function `Operation` interprets primitive operations of the target language: for each primitive operation encountered during the interpretation of a program, function `Operation` checks the types of the arguments and performs computation on the more refined primitive operation used by the interpreter.

Given the following factorial program (written in the language to be interpreted

```
datatype Optr = Plus | Times | Negate | Equal
datatype Exp = C of Univ | V of Univ | Opr of Optr * list
             | Condition of Exp * Exp * Exp
             | Apply of Univ * list
datatype FDef =  FD of Univ * list * Exp
Fun Intpr (progm ins)
    Filter (Specialize (Par Par))
    (let ( [p (Sel progm 1)] [body (Sel p 3)] [vars (Sel p 2)] )
     (Eval body (updenv (nil) vars ins) progm))
Fun Eval (A env main) Void
    caseType A
    (C => (Sel A 1)
     V => (envLookup env (Sel A 1))
     Opr => (let ( [op (Sel A 1)] [es (Sel A 2)] )
              (Operation op (Meval es env main)) )
     Condition => (let ( [e1 (Sel A 1)] [c (Eval e1 env main)]
                         [e2 (Sel A 2)] [e3 (Sel A 3)] )
                    (if c (Eval e2 env main) (Eval e3 env main)))
     Apply => (let ( [args (Sel A 2)] [cs (MEval args env main)]
                     [fcn (funLookup main (Sel A 1))] )
                (EvFun fcn cs main) ) )
Fun Meval (args env main) Void
    caseType args
     (Nil => (nil)
      x => (Cons (Eval (Sel args 1) env main)
                 (Meval (Sel args 2) env main)) )
Fun EvFun (fcn vs main)
          Filter(Specialize (Par Par Par))
    (let ( [vars (Sel fcn 2)] [e (Sel fcn 3)]
           [env (updenv (nil) vars vs)] )
      (Eval e env main))
```

Figure 4.3: Interpreter for Untyped Language — Part 1

```
Fun Operation (Op vs) Void
  caseType op
  (Plus => (let ([v1 (Sel vs 1)] [v2 (Sel (Sel vs 2) 1)] )
             (if (and (isInt v1) (isInt v2))
                 (addInt v1 v2)
                 "Error: Plus with non integer args"))
   Times => (let ([v1 (Sel vs 1)] [v2 (Sel (Sel vs 2) 1)] )
              (if (and (isInt v1) (isInt v2))
                  (mulInt v1 v2)
                  "Error: Times with non integer args"))
   Equal => (let ([v1 (Sel vs 1)] [v2 (Sel (Sel vs 2) 1)] )
              (if (and (isInt v1) (isInt v2))
                  (eqInt v1 v2)
                  (if (and (isStr v1) (isStr v2))
                      (eqStr v1 v2)
                      "Error: Equal with non integer args")))
   Negate =>  (neg (Sel vs 1)) )
```

Figure 4.4: Interpreter for Untyped Language — Part 2

by the interpreter) as an input to the interpreter,

```
[(FD "F" ["X"]
    (Condition (Opr (Equal) [(V "X"),(C 0)])
       (C 1)
       (Opr (Times)
         [(V "X"),
          (Apply "F" [(Opr (Plus)
                           [(V "X"),(Opr (Negate) [(C 1)])])])])))]
```

the residual program produced by conventional partial evaluation of the interpreter with respect to the factorial program and dynamic program input is shown below. Notice that type-checking code is in-lined. (These are marked with symbol **%%** in the residual program.)

```
Fun EVFUN2 (x)
    (If (If (ISINT x)                                          %%
            (EQINT x 0)
```

```
        "Error: Equal with non integer args")
    1
    (Let ( [V2 (EVFUN2
                (If (ISINT x)                                    %%
                    (ADDINT x ~1)
                    "Error: Plus with non integer args"))] )
      (If (AND (ISINT x) (ISINT V2))                             %%
          (MULINT x V2)
          "Error: Times with non integer args")))
```

PPE can remove type-checking code using the *type facet*. The facet domain is a flat lattice consisting of the set of all type names (such as `Int`, `Bool`, *etc.*). The facet operators are all the refined primitive operators used by the interpreter. To remove type-checking code, we specialize the interpreter with respect to the factorial program *and* the program input of type `Int` (but dynamic value). The result is shown below. Notice that all interpretive overhead has been removed.

```
Fun EVFUN2 (x)
    (If (EQINT x 0)
        1
        (MULINT x (EVFUN2 (ADDINT x ~1))))
```

A similar result can be obtained using systems such as Redfun and Fuse [Weise *et al.*, 1991]. Both systems are designed to specialize untyped programs, rather than typed programs. Therefore, they have the type information built in. They lack the flexibility to freely introduce and remove static information into their systems. (This disadvantage was discussed at the end of Section 4.2.1.)

4.2.3 Other Applications

Parameterizing partial evaluation opens up new areas of application of partial evaluation. Most notably, it increases the opportunity for producing compiled programs with different kinds of optimization.

We have seen how typing-checking code can be removed from the residual program using PPE. A similar technique can be applied to produce other optimized

compiled programs (*i.e.*, residual programs) using other kinds of static information. For instance, we have implemented a facet that captures information about whether an integer is a bignum or a fixnum. Using this information, the interpreter is able to choose at partial-evaluation time the appropriate operations for integers.

Static properties can also represent operational information. For instance, it is possible to define a facet that captures information about single-threadedness of vectors.[2] This information again allows the interpreter to choose the appropriate primitive operations for vectors.

All these examples require the interpreter to recognize and process some operational information. Thus, the interpreter is no longer the standard one; rather, it is a non-standard interpreter that includes these operational behaviors. Nevertheless, the partial evaluator itself is transparent to these different interpreters.

The notions of facets and products of facets provide a natural way to model static properties used in the compilation phase. They provide a common backbone whereby compiler and partial evaluator interact. This may enhance the role of partial evaluation in compilation. Experiment in this area is underway.

In conventional partial evaluation, we are able to obtain a dedicated residual program for formatting output, by specializing a general formatting program with respect to a specific format. Using PPE, we create even more opportunities for generating dedicated residual programs. A potential application would be to capture the static properties of a class of devices into facets, and have a parameterized partial evaluator specialized a general device driver routine with respect to that class of devices. This may produce a more dedicated driver that only handles that class of devices.

[2]In functional language, updating a vector usually creates a copy of the entire new vector. If a vector being updated has no other references to it at the time of the update, then the update could be done *destructively*, effectively re-using the old vector, and thus achieving the same efficiency as an imperative assignment to the vector. A vector having this property is said to be *single-threaded* [Hudak and Bloss, 1985,Hudak, 1987].

Program	PPE(sec)	PE(sec)
Inner-product (n=20)	0.48	0.47
Pattern matcher	4.14	11.12
Compiling Inner-Product	7.99	8.62
Compiling using type info	2.78	2.52

Table 4.1: Partial-Evaluation Time Measurement

4.2.4 Performance Measurement

So far, we have seen the qualitative improvement of the residual programs produced by PPE. We now look at the quantitative aspect of PPE; our objective in this section is to measure the time taken for performing PPE.

Without user-defined facets, a parameterized partial evaluator behaves like a conventional partial evaluator. In fact, it differs from the conventional case only in that it looks up the definition of primitive operators from a list of products of facets (with each product contains only the partial-evaluation facet) instead of from the environment. Since the optimization techniques implemented in a conventional partial evaluator can also be applied to a parameterized partial evaluator, we can safely replace a conventional partial evaluator by a parameterized one without user-defined facets. The advantage of comparing PPE with and without facets is that both partial evaluations are treated with the same kind of optimizations. Therefore, the result of comparison shows the overhead of introducing facets into the systems, rather than other irrelevant factors.

Table 4.1 shows the time taken for generating residual programs through partial evaluation with and without facets. The second column lists the partial-evaluation time taken to do the job with facets, while the third column lists the time taken to do it without facets.

The first test program is the inner-product program that we discussed in chapter 2. The vector size used in the test is 20. The facet used is the vector-size facet defined in Section 4.1.1. In the case of partial evaluation without facet, we modify the program so that the main function (`iprod`) now has one more argument — vector

Program	Facet Analysis(sec)	BTA(sec)
Inner-product	0.55	0.01
Pattern matcher	11.45	0.17
Compiling Inner-Product	243.55	1.85

Table 4.2: Analysis Time Measurement

size. In doing so, we are able to pass size information as static value.

The second row in the table shows the time taken to specialize a pattern matcher with respect to a specific pattern string and a partially-known subject string, as was described in Section 4.2.1. We have already seen the qualitative improvement in using facets during partial evaluation; here, we show that there is also an improvement in partial-evaluation time. This is because, by using a possible-value facet, a lot of conditional tests in the program can be reduced to constants, and so only *one* conditional branch needs to be partially evaluated. In conventional partial evaluation, many conditional tests cannot be reduced to constants, and *both* branches of the conditional need to be partially evaluated.

In the third test, we specialize an interpreter with respect to the inner-product program and vector-size information. In the final test, we determine the time taken to specialize an interpreter with respect to a factorial program and type information.

Except in the second test, the other tests show little difference in partial-evaluation time. Besides the fact that facet operations used in these tests are not computationally intensive, another less obvious reason is that with PPE, the user can select only those facets that are of interest for current partial evaluation; this reduces the time taken for computations over useless facets.

Certainly, the user can introduce facets with expensive computations, and consequently PPE using that facet can become time consuming. However, from the measurements provided above, we see that facet computations shorten the partial-evaluation time when they can reduce conditional tests to constants. This observation should be a good guideline for selecting facets for PPE.

Similar timing results (*i.e.*, same as that depicted in Table 4.1) can be obtained for

off-line specialization. However, facet analysis will in general run more slowly than binding-time analysis. Timing comparison for facet analysis and binding-time analysis is shown in Table 4.2. Two reasons account for the slow-down in facet analysis: (1) Primitive operations are handled uniformly in binding-time analysis regardless of their types. (That is, when the arguments of a primitive operation are of value *static*, its result is *static*; otherwise, it is *dynamic*.) This avoids looking into the environment for primitive-operation definition. On the other hand, every primitive operation in facet analysis involves looking into the list of products of abstract facets for its definition. This slow down is proportional to the length of the list of products of abstract facets. (2) Totally-static data structures in facet analysis are not represented by value *static*; instead, they are represented by similar structures with static components replaced by product-of-abstract-facet values. This implies that more time is required for facet analysis (than for binding-time analysis) to construct and dereference totally-static data structures. The time difference can become prominent when specializing an interpreter, where the program to be interpreted is a totally-static data structure. This result is shown in the third row of Table 4.2.

4.3 Discussion

We begin the implementation of both on-line and off-line PPE by translating the semantic specifications given in Chapter 3 into Standard ML programs. We then apply several optimizations to make them efficient. These optimizations have already been used in conventional partial evaluation. Generally, the quality of the residual programs produced by PPE is better than that produced by conventional partial evaluation. Furthermore, since none of the facets is built in, only those facets pertaining to an application are used during partial evaluation. This can eliminate many computations over useless static properties, which is not possible in those partial evaluators that have built-in static properties.

In terms of time efficiency, the performance of on-line PPE (and off-line special-

ization) depends partly on the amount of facet computation required. However, in partially evaluating a conditional expression, facet computation can reduce a conditional test to a constant, so that only one conditional branch is required for partial evaluation; this consequently reduces partial-evaluation time.

Facet analysis tends to run more slowly than binding-time analysis because totally-static data structures are represented fully using data structures and products of abstract facets, instead of value *static*. A possible way to reduce analysis time would be to include constants in the binding-time domain, as was described in Section 2.5.2. In doing so, we provide an opportunity for reducing conditional tests to constants; thus, facet analysis needs to analyze only one conditional branch, instead of two. This technique is used in on-line PPE. Again, we notice that optimization technique that is developed at the on-line level is transferred to the off-line level.

In Section 4.2.2, we show the technique of converting a parameterized partial evaluator that specializes typed programs to one that specializes untyped programs. This implies that the framework of PPE is applicable to both typed and untyped languages. When dealing with an untyped language, a universal algebra is defined containing a universal domain and all primitive operations. Facets become abstractions of this universal algebra. Since all facets are derived from the same semantic algebra, there is only one product of facets. The resulting product domain has the similar structure as the type domain used by the type evaluator in [Young and O'Keefe, 1988].

Chapter 5

Conclusion

In this chapter, we briefly discuss related work and future work, and sum up the thesis.

5.1 Related Work

Redfun [Haraldsson, 1977] 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.,* [Schooler, 1984,Guzowski, 1988,Berlin, 1990]). The latest system developed along this line is Fuse [Weise and Ruf, 1990,Weise *et al.*, 1991]. 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 the 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. Also, they do not provide a safe and systematic approach to introduce user-defined static properties; when a static property is introduced, it 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 [Futamura and Nogi, 1988,Takano, 1991] is a program optimization approach that aims at specializing programs with respect to various information (called *u-information*) such as logical structure of programs, axioms for abstract data types, etc. 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 thus 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 [Takano, 1991]. Generalized Partial Computation does not address off-line partial evaluation. Also the literature in the field [Sestoft, 1990] does not report any implementation of this approach.

Lastly, in developing the correctness properties of PPE, several ideas in denotational semantics have influenced our work:

1. Collecting interpretation [Hudak and Young, 1991,Jones and Mycroft, 1986]. This idea enables the specification and correctness proof of polyvariant specialization.

2. Factorized semantics [Jones and Nielson, 1990]. This idea leads to the definition of formal relationship of on-line and off-line PPE. It also opens up the opportunity for the transfer of techniques developed at either strategy to another.

3. Satisfiability criteria [Gomard, 1992]. This technique forms the basic foundation for proving the correctness of residual expressions generated by partial evaluation.

5.2 Future Work

The concept of PPE advances the research in partial evaluation to a new frontier. Not only does it extend conventional ideas about partial evaluation in a natural way,

but it also provides the users with the opportunity to steer the partial-evaluation process via various static properties pertaining to the application at hand. The work presented in this thesis can be extended in several directions, many of which have been discussed at the end of each chapter. Below, we list other possible extensions.

1. **Funneling facet information.** In the specification of on-line partial evaluation given in Figures 3.8 and 3.9, we provided a less complete treatment of conditional expression than the one described in Redfun [Haraldsson, 1977].[1] Indeed, Redfun is able to extract more refined properties from the conditional test and funnel these properties and their complements to the consequent and alternate branches respectively.

   By defining a facet domain as a complete lattice, we can easily extract more refined properties from the conditional test. However, we have difficulty in producing the complement of these properties as accurate as desired, since a lattice element may have more than one (and sometimes no) complement [Davey and Priestley, 1990].

   This problem is common in the field of functional languages. Shivers, in describing the technique for performing type recovery in Scheme [Shivers, 1991], provides a solution to the funneling problem for type information. It may be possible to generalize the technique to handle arbitrary facets.

2. **Polyvariant facet analysis.** In the specification of off-line PPE, we described a monovariant facet analysis. In fact, the result of facet analysis can be tremendously improved by introducing polyvariant facet analysis. This is a natural extension of polyvariant binding-time analysis. The latter has been used in the off-line partial evaluator Schism to successfully specialize a functional version of Prolog interpreter [Consel and Khoo, 1991b]. However, preliminary result shows that although the analysis result can be extremely accurate, it tends to be slow. Much work remains to be done in this area.

[1] However, the current treatment is commonly practiced in most conventional partial evaluators.

3. **Extending language domain.** Although this thesis describes PPE for a first-order functional language, the framework as it is can be used for partially evaluating programs written in other languages, such as higher order language or imperative language.

   In the case of higher order functional language, a sketch for PPE of higher order programs was presented in [Consel and Khoo, 1991a]. Future work is needed to formally specify and prove the correctness of PPE of higher order programs. This will involve using closure analysis [Sestoft, 1989] in the off-line analysis phase, and correspondingly, closure evaluation in the on-line PPE phase. Preliminary studies in this direction indicate that such an extension is feasible since it would be based on an existing framework for abstract interpretation of higher-order programs, such as [Jones, 1991].

5.3 Conclusion

This thesis aims at investigating a general and formal treatment of partial evaluation using static properties, with the hope of developing a formal framework for partial evaluation such that static properties can be introduced safely and uniformly into both on-line and off-line strategies. We presented a generic form of partial evaluation, called parameterized partial evaluation, that goes beyond the objective of investigation in that it also solves some open issues such as the relationship between on-line and off-line partial evaluation and the correctness proof of polyvariant specialization.

On the practical side, by providing different facets for different applications, parameterized partial evaluation was shown to provide a framework for specifying and implementing a large family of partial evaluators, each being dedicated to a specific application. Besides our implementation, parameterized partial evaluation has already been successfully implemented for a first order subset of ML at CMU [Colby and Lee, 1991].

Bibliography

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

[Abramsky, 1990] A. Abramsky. Abstract Interpretation, Logical Relations and Kan Extensions. *Logic and Computation*, 1(1):5–40, 1990.

[Beckman *et al.*, 1976] L. Beckman, A. Haraldsson, O. Oskarsson, and E. Sandewall. A Partial Evaluator, and Its Use as a Programming Tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[Berlin, 1990] A. Berlin. Partial Evaluation Applied to Numerical Computation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 139–150. ACM, 1990.

[Bondorf and Danvy, 1991] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[Bondorf *et al.*, 1988] A. Bondorf, N. D. Jones, T. Mogensen, and P. Sestoft. Binding Time Analysis and the Taming of Self-Application. Diku report, University of Copenhagen, Copenhagen, Denmark, 1988.

[Bondorf, 1988] A. Bondorf. Pattern Matching in a Self-Applicable Partial Evaluator. Diku report, University of Copenhagen, Copenhagen, Denmark, 1988.

[Bondorf, 1990] A. Bondorf. *Self-Applicable Partial Evaluator.* PhD thesis, University of Copenhagen, DIKU, Copenhagen, Denmark, 1990.

[Colby and Lee, 1991] C. Colby and P. Lee. An Implementation of Parameterized Partial Evaluation. *Bigre Journal*, 74:82–89, 1991.

[Consel and Danvy, 1989] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30(2):79–86, 1989.

[Consel and Danvy, 1990] C. Consel and O. Danvy. From Interpreting to Compiling Binding Times. In N. D. Jones, editor, *Proceedings of the European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.

[Consel and Khoo, 1991a] C. Consel and S. C. Khoo. Parameterized Partial Evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–106, 1991.

[Consel and Khoo, 1991b] C. Consel and S. C. Khoo. Semantics-Directed Generation of a Prolog Compiler. In J.Maluszyński and M. Wirsing, editors, *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, pages 135–146. Springer-Verlag, 1991.

[Consel and Khoo, 1992] C. Consel and S. C. Khoo. On-line & Off-line Partial Evaluation: Semantic Specifications and Correctness Proofs. Research Report 912, Yale University, New Haven, Connecticut, USA, 1992.

[Consel, 1988] C. Consel. New Insights into Partial Evaluation: the Schism Experiment. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes*

in Computer Science, pages 236–246. Springer-Verlag, 1988.

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

[Consel, 1990a] C. Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 264–272. ACM, 1990.

[Consel, 1990b] C. Consel. *The Schism Manual.* Yale University, New Haven, Connecticut, USA, 1990. Version 1.0.

[Cooper *et al.*, 1992] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure Cloning. In *Proceedings of the 1992 International Conference on Computer Languages.* IEEE Computer Society Press, 1992.

[Cousot and Cousot, 1977] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Danvy and Filinski, 1991] O. Danvy and A. Filinski. Representing Control, A Study of the CPS Transformation. Technical Report CIS-91-2, Kansas State University, Manhattan, Kansas, USA, 1991.

[Danvy, 1991] O. Danvy. Semantics-Directed Compilation of Non-Linear Patterns. *Information Processing Letters*, 37:315–322, March 1991.

[Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[Emanuelson and Haraldsson, 1980] P. Emanuelson and A. Haraldsson. On Compiling Embedded Languages in LISP. In *1980 Lisp Conference, Stanford, California*, pages 208–215, 1980.

[Emanuelson, 1980] P. Emanuelson. *Performance Enhancement in a Well-Structured Pattern Matcher Through Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1980. Linköping Studies in Science and Technology Dissertations N° 55.

[Futamura and Nogi, 1988] Y. Futamura and K. Nogi. Generalized Partial Computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[Futamura, 1971] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[Gomard, 1992] C. K. Gomard. A Self-Applicable Partial Evaluator for the Lambda-Calculus: Correctness and Pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[Guzowski, 1988] 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.

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

[Hudak and Bloss, 1985] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the ACM Symposium*

on Principles of Programming Languages, pages 300–314, 1985.

[Hudak and Young, 1991] P. Hudak and J. Young. Collecting Interpretations of Expressions. *ACM Transactions on Programming Languages and Systems*, 13(2):269–290, 1991.

[Hudak, 1987] P. Hudak. A Semantic Model of Reference Counting and Its Abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[Jones and Muchnick, 1976] N. D. Jones and S. S. Muchnick. Some Thoughts towards the Design of an Ideal Language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 77–94, 1976.

[Jones and Mycroft, 1986] N. D. Jones and A. Mycroft. Data Flow Analysis of Applicative Programs using Minimal Function Graphs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1986.

[Jones and Nielson, 1990] 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.

[Jones *et al.*, 1985] N. D. Jones, P. Sestoft, and H. Søndergaard. An Experiment in Partial Evaluation: the Generation of a Compiler Generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[Jones *et al.*, 1989] 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.

[Jones, 1988] N.D. Jones. Automatic Program Specialization: A Re-Examination from Basic Principles. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[Jones, 1990] N. D. Jones. Partial Evaluation, Self-Application and Types. In M.S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 639–659. Springer-Verlag, 1990.

[Jones, 1991] N. D Jones. A Minimal Function Graph Semantics as a Basis for Abstract Interpretation of Higher Order Programs. *Bigre Journal*, 74, 1991.

[Jørgensen, 1990] J. Jørgensen. Generating a Pattern Matching Compiler by Partial Evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 177–195. Springer-Verlag, August 1990.

[Kishon *et al.*, 1991] A. Kishon, P. Hudak, and C. Consel. Monitoring Semantics: a Formal Framework for Specifying, Implementing and Reasoning about Execution Monitors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, 1991.

[Kishon, 1992] A. Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring.* PhD thesis, Yale University, New Haven, Connecticut, 1992. Report No. YALEU/DCS/RR-905.

[Kleene, 1952] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

[Launchbury, 1990] J. Launchbury. *Projection Factorisation in Partial Evaluation.* PhD thesis, Department

of Computing Science, University of Glasgow, Scotland, 1990.

[Lombardi and Raphael, 1964] L. A. Lombardi and B. Raphael. Lisp as the Language for an Incremental Computer. In E. C. Berkeley and D. G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219. MIT Press, Cambridge, Massachusetts, 1964.

[Lombardi, 1967] L. A. Lombardi. Incremental Computation. In F. L. alt and M. Rubinoff, editors, *Advances in Computers*, volume 8, pages 247–333. Academic Press, 1967.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Mizuno and Schmidt, 1990] M. Mizuno and D. Schmidt. A Security Flow Control Algorithm and Its Denotational Semantics Correctness Proof. Technical Report CS-90-21, Kansas State University, Manhattan, Kansas, 1990.

[Mogensen, 1988] T. Mogensen. Partially Static Structures in a Self-applicable Partial Evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–348. North-Holland, 1988.

[Mogensen, 1989] T. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, University of Copenhagen, DIKU, Copenhagen, Denmark, 1989.

[Nielson, 1989] F. Nielson. Two-level Semantics and Abstract Interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[Ruf and Weise, 1991] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Volume 26, No. 9, pages 321–333, 1991.

[Safra, 1990] S. Safra. Partial Evaluation of Concurrent Prolog and Its Implications. Master's thesis, Weizmann Institute of Science, Israel, 1990.

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

[Schooler, 1984] R. Schooler. Partial Evaluation as a Means of Language Extensibility. Master's thesis, M.I.T. (LCS), Massachusetts, U.S.A, 1984.

[Sestoft, 1985] P. Sestoft. The Structure of a Self-Applicable Partial Evaluator. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 236–256, 1985.

[Sestoft, 1988] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[Sestoft, 1989] P. Sestoft. Replacing function parameters by global variables. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53. ACM Press and Addison-Wesley, 1989.

[Sestoft, 1990] P. Sestoft. Annotated Bibliography on Partial Evaluation and Mixed Computation. Diku report, University of Copenhagen, Copenhagen, Denmark, 1990.

[Shivers, 1991] O. Shivers. Data-Flow Analysis and Type Recovery in Scheme. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 47–88. MIT Press, 1991.

[Steele, 1978] G. L. Steele. Rabbit: A Compiler for Scheme. Master's thesis, M.I.T (A.I. LAB.), Massachusetts, U.S.A, 1978.

[Sundaresh and Hudak, 1991] R. S. Sundaresh and P. Hudak. Incremental Computation via Partial Evaluation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 1–13, 1991.

[Sundaresh, 1991] R. S. Sundaresh. *Incremental Computation via Partial Evaluation.* PhD thesis, Yale University, New Haven, Connecticut, 1991. Report No. YALEU/DCS/RR-889.

[Takano, 1991] A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Volume 26, No. 9, pages 1–11, 1991.

[Weise and Ruf, 1990] D. Weise and E. Ruf. Computing Types During Program Specialization. Technical Report 441, Stanford University, Stanford, USA, 1990.

[Weise *et al.*, 1991] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic Online Partial Evaluation. In J. Hughes, editor, *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191. Springer-Verlag, 1991.

[Young and O'Keefe, 1988] J. Young and P. O'Keefe. Experience with A Type Evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North-Holland, 1988.