A Modular Implementation of Partial Evaluation

Christopher Colby Peter Lee

March 1992

CMU-CS-92-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Charles Consel and Siau Cheng Khoo have developed a technique for parameterizing partial evaluation and binding-time analysis with respect to abstract domains [5]. We have found the modules system of Standard ML [12, 15] to be a useful vehicle for implementing a similar parameterization technique. Furthermore, the ability to parameterize binding-time analysis indicates that the technique may be useful for implementations of collecting interpretations in general. This paper describes our implementation of parameterized partial evaluation, with a particular focus on the use of the Standard ML modules system.

This research was partially supported by the National Science Foundation under grant #CCR-9057567 and in part by the NSF Graduate Research Fellowship Program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

Source program:

```
factiter(n,r) = if eq(n,0)
                   1
                   factiter(minus(n,1),times(n,r))
```

Input $\langle 5, 1 \rangle$ yields the residual program:

```
factiter7() = 120
```

Input $\langle 5, \top \rangle$ yields the residual program:

```
factiter7(r) = times(1,times(2,times(3,times(4,times(5,r)))))
```

Figure 1: Examples of partial evaluation.

1. Introduction

A partial evaluator, when given a program and a partial specification of the program's input data, produces a residual program that takes the remainder of the input and computes the proper result. As an example of partial evaluation, see Figure 1. Several pragmatic considerations lead to difficulties in the development of a partial evaluator. The first consideration is the efficiency of the residual programs. Ideally, the availability of the partial input should enable many optimizations. However, this is complicated by the desire to handle programs that use higher-order functions [8] and complex data structures [13]. A second consideration is self-application—the ability to partially evaluate the partial evaluator [10]. Self-applicable partial evaluators can be useful in compiler generation [7] as well as other areas, but in practice they require a two-pass strategy involving a binding-time analysis [11]. Despite these difficulties, self-applicable partial evaluators such as Similix [2] and Schism [3] have been developed and used in nontrivial applications.

A third consideration that has received far less attention has to do with the method for specifying the partial inputs. The simplest approach allows individual inputs to be completely specified or unspecified. But a more general notion of partial input can also be useful. For instance, one may want to specify that an input is an unknown, but positive, integer; another possibility might be to specify that an input is a list containing three (unknown) elements; and so on. Such flexibility in the specification of partial input allows more information to be provided to the partial evaluator, thereby leading to more efficient residual programs. For some examples, see Figure 12.

In order to achieve such generality in the specification of partial input, Charles Consel and Siau Cheng Khoo have developed a method for parameterizing partial evaluation and binding-time analysis with respect to abstract domains [5]. Using their method as a general guide, we have used the modules system of Standard ML [12, 15] to implement a simple partial evaluator that is parameterized by modular implementations of abstract interpretations. The parameterization allows a standard implementation of a partial evaluator to be extended in a modular fashion, by abstract domains, thereby leading to flexibility in the specification of the partial input. We have also implemented a binding-time analyzer that is parameterized in the same manner. This leads us to believe that collecting interpretations in general might benefit from this modularity.

We begin this paper with some background about the technique developed by Consel and Khoo, and discuss simple (*i.e.*, not parameterized) partial evaluation. Then, we present a description and an implementation in Standard ML of our system for specifying and combining abstract interpretations. We then describe a partial evaluator written in Standard ML that is parameterized with respect to such abstract interpretations. Finally, we describe an implementation of a binding-time analyzer parameterized in the same way and conclude with future directions of this research.

2. Background

One of the key notions used by Consel and Khoo is the *facet*. Given a semantic algebra, a facet defines an abstraction (or approximation) of that algebra. A semantic algebra $[\mathbf{D}; O]$ consists of a semantic domain $\mathbf{D}$ and a set of primitive operations O on this domain. A facet for that algebra is an algebra $[\widehat{\mathbf{D}}; \widehat{O}]$ defined by an abstraction funtion $\hat{\alpha} : \mathbf{D} \to \widehat{\mathbf{D}}$, where

1. $\widehat{\mathbf{D}}$ is an algebraic lattice of finite height.
2. If $p \in O$ is a closed operator (*i.e.*, $p : \mathbf{D}^* \to \mathbf{D}$), then $\hat{p} : \widehat{\mathbf{D}}^* \to \widehat{\mathbf{D}}$ is its corresponding abstract version in $\widehat{O}$, and $\hat{\alpha} \circ p \sqsubseteq \hat{p} \circ \hat{\alpha}^*$.
3. If $p \in O$ is an open operator with functionality $\mathbf{D}^* \to \mathbf{D}'$, where $\mathbf{D}'$ is some domain different from $\mathbf{D}$, then $\hat{p} : \widehat{\mathbf{D}}^* \to \widehat{\mathbf{Const}}$ is its corresponding abstract version in $\widehat{O}$, and $\hat{\tau} \circ p \sqsubseteq \hat{p} \circ \hat{\alpha}^*$, where $\hat{\tau} : \mathbf{D}' \to \widehat{\mathbf{Const}}$ is a function that maps domain elements into the lifted domain of basic values.

An example facet is the rule-of-signs abstraction for the semantic algebra $[\mathbf{Int}_\perp; \{+, <\}]$ where $\widehat{\mathbf{D}} = \{\perp, \text{Pos}, \text{Zero}, \text{Neg}, \top\}$, $\hat{+} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \to \widehat{\mathbf{D}}$ is the closed $+$ operator defined over signs and yielding signs (*e.g.*, $\hat{+}\langle\text{Pos}, \text{Pos}\rangle = \text{Pos}$), and $\hat{<} : \widehat{\mathbf{D}} \times \widehat{\mathbf{D}} \to \widehat{\mathbf{Const}}$ is the open $<$ operator that may yield concrete values (*e.g.*, $\hat{<}\langle\text{Neg}, \text{Pos}\rangle = [\![\text{true}]\!]$).

Consel and Khoo furthermore define the *product of facets*, which combines a set of facets defined for a given semantic algebra. Given facets $[\widehat{\mathbf{D}}_i; \widehat{O}_i]$ for $i \in \{1 \ldots m\}$ defined for the semantic algebra $[\mathbf{D}; O]$, one can define the product of facets, denoted $[\widehat{\mathcal{D}}; \widehat{\Omega}]$, where $\widehat{\mathcal{D}}$ is, roughly speaking, a kind of product of the domains $\widehat{\mathbf{D}}_i$, and where there is a *product operator* $\omega_p \in \widehat{\Omega}$ for each operator $p \in \mathbf{D}$. If p is a closed operator, then $\omega_p : \widehat{\mathcal{D}}^* \to \widehat{\mathcal{D}}$. If p is an open operator, then $\omega_p : \widehat{\mathcal{D}}^* \to \widehat{\mathbf{Const}}$. Intuitively, ω_p triggers each facet operator $\hat{p}_i$ with its corresponding abstract values from the products.

Further details of this framework are presented in [5], which demonstrates as examples a simple online partial evaluator and a binding-time analyzer for a first-order functional language, both of which are parameterized with respect to a product of facets. The binding-time analysis involves a notion of *facet analysis* and *abstract facets* to express the collecting interpretation.

We have developed a system embodying notions similar in spirit to the facets, product of facets, and abstract facets of Consel and Khoo. Furthermore, we have implemented this system in the language Standard ML in a way that makes good use of ML's modules system. We describe this implementation in Section 4.

3. A Simple Online Partial Evaluator

Before we present our system of parameterization, we shall first describe a semantics of a simple online partial evaluator. Figure 2 is a semantics of a partial evaluator for a strict first-order functional language similar to the one presented in [5] and other papers. The APP function, which is not shown in Figure 2, makes decisions about when to specialize functions. When a function is specialized, either the result is inlined or a call to a new specialized function (which is added to the program returned by APP) is returned.

In this simple online evaluator, the program is symbolically evaluated. Applications of primitive operations are checked to see if all of the arguments are constant, and if so the entire application is "constant-folded" away. Hence, constant folding is built into the partial evaluator. It is "hard-coded" into the $\mathcal{K}_p$ function. As described in Section 1, however, we would like to have more flexibility in expressing the values of an expression during partial evaluation (*e.g.*, POS representing a positive integer). We will thus see that primitive operations will have to be handled differently.

In our system, instead of building a $\mathcal{K}_p$ function in the semantics, we supply the partial evalutator with a *facet*, which defines the abstract domains over which values are taken and the definitions of the primitive operations over these domains. A facet is thus a collection of abstract interpretations. One defines a single such abstract interpretation by a *facet specification*, and one may combine multiple facets into a single facet. The standard notion of constant folding can be encapsulated in a facet, as well; as we shall describe in Section 5, one may achieve this by writing a facet specification similar to the $\mathcal{K}_p$ function of Figure 2. But facet specifications are expressive enough to encapsulate many different abstract interpretations. Later, we shall present a semantics for a partial evaluator for our first-order functional language that is parameterized with respect to a facet. But first, we shall explain facets themselves.

4. Facet Specifications, Facets, and the Combinations of Facets

First we shall present some preliminaries. The usual notion of the lift operation involves adding a $\perp$ element to a set of values. Our "lift", however, adds $\top$ as well. In Standard ML code, we have the `LIFTED` signature in Figure 3 which specifies a lifted domain. Type `T` represents the domain to be lifted. The signature provides equality, join, and meet operations for the lifted domain. (The equality operation is provided by Standard ML's polymorphic equality and enforced by the `eqtype` requirement for `T`.) The signature itself does not enforce the ordering shown in the lattice diagram, but it is assumed that functions `join` and `meet` will be defined appropriately.

The `Lift` functor in Figure 3 takes a type representing a set of values and produces a flat lifted domain of that type. Also shown is the result of applying the functor to the set of constants of our first-order language. We denote the resulting lattice $\widehat{\mathbf{Const}}$. As we shall see, $\widehat{\mathbf{Const}}$ will play a major role in our parameterized partial evaluator.

Now, we are ready to present facets as encapsulations of abstract interpretations. One specifies a facet with a *facet specification*. The components of a facet specification and the associated ML signature `FACET_SPECIFICATION` are shown in Figure 4. Notice that α and ι, while monotonic,

- Syntactic Domains

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

$$\begin{array}{rcl} e & ::= & c \mid x \mid p(e_1,\ldots,e_n) \mid f(e_1,\ldots,e_n) \mid \text{if}\, e_1\, e_2\, e_3 \\ \sigma & ::= & \{f_i(x_1,\ldots,x_{n_i}) = e_i\} \ \ (f_1 \text{ is the main function}) \end{array}$$

- Semantic Domains

$$\rho \in \textbf{Env} = \textbf{Var} \rightarrow \textbf{Exp}$$

- Valuation Functions

$$\begin{array}{rcl} \mathcal{P} & : & \textbf{Prog} \rightarrow \textbf{Exp}^* \rightarrow \textbf{Prog} \\ \mathcal{E} & : & \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{Prog} \rightarrow \textbf{Exp} \times \textbf{Prog} \\ \mathcal{K}_p & : & \textbf{Primop} \rightarrow \textbf{Const}^* \rightarrow \textbf{Const} \\ \text{APP} & : & \textbf{Fn} \rightarrow \textbf{Exp}^* \rightarrow \textbf{Env} \rightarrow \textbf{Program} \rightarrow \textbf{Exp} \times \textbf{Program} \end{array}$$

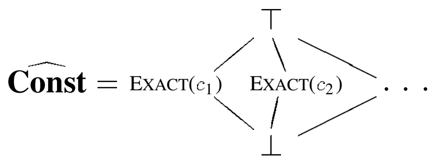
$$\begin{array}{rcl} \mathcal{P}\,\sigma\,\langle e'_1,\ldots,e'_{n_1}\rangle & = & (\mathcal{E}[\![f_1(x_1,\ldots,x_{n_1})]\!](\bot[e'_k/x_k])\sigma)\downarrow_2 \\ & & \text{where } \ \sigma = [\![\{f_i(x_1,\ldots,x_{n_i}) = e_i\}]\!] \end{array}$$

$$\begin{array}{rcl} \mathcal{E}\,[\![c]\!]\,\rho\,\sigma & = & \langle [\![c]\!], \sigma\rangle \\ \mathcal{E}\,[\![x]\!]\,\rho\,\sigma & = & \langle \rho[\![x]\!], \sigma\rangle \\ \mathcal{E}\,[\![p(e_1,\ldots,e_n)]\!]\,\rho\,\sigma & = & \bigwedge_{i=1}^{n}(e'_i \in \textbf{Const}) \rightarrow \langle \mathcal{K}_p[\![p]\!]\langle e'_1,\ldots,e'_n\rangle, \sigma_n\rangle,\ \langle [\![p(e'_1,\ldots,e'_n)]\!], \sigma_n\rangle \\ & & \text{where} \quad \begin{array}{rcl} \langle e'_1, \sigma_1\rangle & = & \mathcal{E}[\![e_1]\!]\rho\sigma \\ & \vdots & \\ \langle e'_n, \sigma_n\rangle & = & \mathcal{E}[\![e_n]\!]\rho\sigma_{n-1} \end{array} \\ \mathcal{E}\,[\![\text{if}\ e_1\ e_2\ e_3]\!]\,\rho\,\sigma & = & (e'_1 \in \textbf{Const}) \rightarrow \\ & & \quad (e'_1 \rightarrow \mathcal{E}[\![e_2]\!]\rho\sigma_1,\ \mathcal{E}[\![e_3]\!]\rho\sigma_1), \\ & & \quad \langle [\![\text{if}\ e'_1\ e'_2\ e'_3]\!], \sigma_3\rangle \\ & & \quad \text{where} \ \begin{array}{rcl} \langle e'_2, \sigma_2\rangle & = & \mathcal{E}[\![e_2]\!]\rho\sigma_1 \\ \langle e'_3, \sigma_3\rangle & = & \mathcal{E}[\![e_3]\!]\rho\sigma_2 \end{array} \\ & & \text{where } \ \langle e'_1, \sigma_1\rangle = \mathcal{E}[\![e_1]\!]\rho\sigma \\ \mathcal{E}\,[\![f(e_1,\ldots,e_n)]\!]\,\rho\,\sigma & = & \text{APP}[\![f]\!]\langle e'_1,\ldots,e'_n\rangle\rho\sigma_n \\ & & \text{where} \quad \begin{array}{rcl} \langle e'_1, \sigma_1\rangle & = & \mathcal{E}[\![e_1]\!]\rho\sigma \\ & \vdots & \\ \langle e'_n, \sigma_n\rangle & = & \mathcal{E}[\![e_n]\!]\rho\sigma_{n-1} \end{array} \end{array}$$

$$\begin{array}{rcl} \mathcal{K}_p\,[\![+]\!]\,\langle c_1, c_2\rangle & = & c_1 + c_2 \\ \mathcal{K}_p\,[\![-]\!]\,\langle c_1, c_2\rangle & = & c_1 - c_2 \\ & \vdots & \end{array}$$

Figure 2: A simple online partial evaluator for a strict first-order functional language.

```
signature LIFTED =
sig
  eqtype T
  datatype Lifted = Top
                  | Exact of T
                  | Bottom
  val join: Lifted -> Lifted -> Lifted
  val meet: Lifted -> Lifted -> Lifted
end
```




```
functor Lift(eqtype T): LIFTED =
struct
  type T = T
  datatype Lifted = Top
                  | Exact of T
                  | Bottom

  fun join Bottom x = x
    | join x Bottom = x
    | join x y = if x=y then x
                 else Top

  fun meet Top x = x
    | meet x Top = x
    | meet x y = if x=y then x
                 else Bottom
end
```

Figure 3: A functor to produce a flat lifted domain and the result of lifting the set of constants.

1. A *concrete domain* $\mathbf{D}$ that must be lifted (as shown in Figure 3).

2. An *abstract domain* $\widehat{\mathbf{D}}$.

3. A monotonic *abstraction function* $\alpha : \mathbf{D} \to \widehat{\mathbf{D}}$ and an associated monotonic *injection function* $\iota : \widehat{\mathbf{D}} \to \mathbf{D}$ satisfying the safety condition that $id_{\mathbf{D}} \sqsubseteq \iota \circ \alpha$.

4. A set O_c of *closed abstract primitive operations*; an operation $\hat{p} \in O_c$ has functionality $\widehat{\mathbf{D}}^* \to \widehat{\mathbf{D}}$.

5. A set O_o of *open abstract primitive operations*; an operation $\hat{p} \in O_o$ has functionality $\widehat{\mathbf{D}}^* \to \mathbf{D}$.

```
signature FACET_SPECIFICATION =
sig
  structure Op: PRIM
  structure D: LIFTED
  eqtype Dhat
  val join: Dhat -> Dhat -> Dhat
  val abs: D.Lifted -> Dhat
  val inj: Dhat -> D.Lifted
  val closedOp: Op.Prim -> bool
  val openOp: Op.Prim -> bool
  val Oc: Op.Prim -> Dhat list -> Dhat
  val Oo: Op.Prim -> Dhat list -> D.Lifted
end
```

Figure 4: Components of a facet specification and the associated Standard ML signature.

do not define a retraction.

As an example of a facet specification, Figure 5 shows exerpts (`Oc` and `Oo` are incomplete) from a Standard ML functor `SignSpec` that implements the specification for the rule-of-signs abstraction of integers. The functor takes as arguments a structure `Absyn` matching signature `ABSYN` that specifies the abstract syntax of our first-order language and a structure `Const` representing a lifted domain. It yields a structure matching the `FACET_SPECIFICATION` signature of Figure 4. In addition, there is a sharing constraint that requires that the "non-lifted" elements of the domain that `Const` represents be the elements of type `Absyn.Const`. In other words, `Const` represents the $\widehat{\mathbf{Const}}$ of Figure 3.

Finally, we can consider combining a set of such abstract interpretations into a single facet. The form of a facet specification does not lend itself to such combinations. Hence, we introduce a slightly different kind of structure, called a *facet*. The components of a facet, along with the associated signature `FACET`, are shown in Figure 6. Note that, except for a name change from `D` to `FacetValue`, the only difference between the definition of a facet specification and the definition of a facet is that there are no open primitive operations in a facet. This will make it possible to combine two facets into one, as we shall see below.

Figure 7 shows a mechanical process, implemented by the functor `MakeFacet`, to convert a facet specification to a facet. Note the definition of the **FacetValue** domain as $\widehat{\mathbf{D}}_S \times \mathbf{D}_S$. Intuitively, the idea behind this is that the **FacetValue** domain is expressive enough to alleviate the need to differentiate between open and closed primitive operations.

Now, we may combine facets. Figure 8 shows a mechanical process to combine two facets with the same concrete domain and corresponding primitive operations to a single facet. The **FacetValue** domain is the domain cross-product of the two components' **FacetValue** domains. The abstraction function is the pairwise composition of the components' abstractions functions, and the injection function returns the best approximation to the results of both components' injection function.

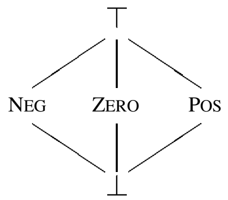
Notice the definition of the abstract primitive operations O. First, the pairwise application of the component facets' corresponding operations is done to yield a compound facet value f. Next, f is injected into $\mathbf{D}$ with ι. Then, if the result of the injection is $\bot_{\mathbf{D}}$, we return the abstraction of $\bot_{\mathbf{D}}$, which is $\bot_{\mathbf{FacetValue}}$. Otherwise, if the result of the injection is $\top_{\mathbf{D}}$, then we simply return f, since it is the most accurate facet value information available. Otherwise, f has injected into a "concrete" value, and we return the abstraction of that value, which is the most accurate facet value possible of such a concrete value. The intuitive result is that when many facets are combined into one, if some abstract interpretation yields a concrete value as a result of a primitive operation (*e.g.*, $\widehat{<}\langle \textsc{Neg}, \textsc{Pos}\rangle$), that value is "propagated" down to the other facets to be abstracted via their respective α functions.

We now have the machinery to capture an arbitrary number of abstract interpretations in a single facet. First, we define each abstract interpretation by writing specifications matching the `FACET_SPECIFICATION` signature of Figure 4. Then, we apply the `MakeFacet` functor of Figure 7 to each facet specification in order to obtain a facet. Finally, we combine all the facets into a single facet with multiple applications of the `CombineFacets` functor of Figure 8.

```
functor SignSpec(structure Absyn: ABSYN
                 structure Const: LIFTED
                 sharing type Absyn.Const = Const.T): FACET_SPECIFICATION =
struct
  exception SignSpec

  structure Op = Absyn.Op


  structure D = Const


  datatype Dhat = Top
                | Neg | Zero | Pos
                | Bottom

  fun join Bottom x = x
    | join x Bottom = x
    | join x y = if (x=y) then x else Top


  fun abs D.Bottom = Bottom
    | abs (D.Exact (Absyn.Int n)) =
      if n=0 then Zero
             else if n>0 then Pos else Neg
    | abs _ = Top


  fun inj Zero = D.Exact (Absyn.Int 0)
    | inj _ = D.Top


  fun closedOp Op.PLUS = true
    | closedOp Op.MINUS = true
    | closedOp Op.TIMES = true
    | closedOp _ = false


  fun openOp Op.GT = true
    | openOp Op.EQ = true
    | openOp _ = false


  val True = D.Exact (Absyn.Tr true)
  val False = D.Exact (Absyn.Tr false)


  fun Oc _ [Bottom,_] = Bottom
    | Oc _ [_,Bottom] = Bottom

    | Oc Op.PLUS [Zero,dhat2] = dhat2
    | Oc Op.PLUS [dhat1,Zero] = dhat1
    | Oc Op.PLUS [Pos,Pos] = Pos
    | Oc Op.PLUS [Neg,Neg] = Neg
    | Oc Op.PLUS _ = Top

    | Oc Op.MINUS ...

    | Oc Op.TIMES ...

    | Oc _ _ = raise SignSpec

  fun Oo _ [Bottom,_] = D.Bottom
    | Oo _ [_,Bottom] = D.Bottom

    | Oo Op.GT [Pos,Neg] = True
    | Oo Op.GT [Pos,Zero] = True
    | Oo Op.GT [Zero,Pos] = False
    | Oo Op.GT [Zero,Zero] = False
    | Oo Op.GT [Zero,Neg] = True
    | Oo Op.GT [Neg,Zero] = False
    | Oo Op.GT [Neg,Pos] = False
    | Oo Op.GT _ = D.Top

    | Oo Op.EQ ...

    | Oo _ _ = raise SignSpec
end
```

$$\mathbf{D} = \widehat{\mathbf{Const}}$$

$\widehat{\mathbf{D}} =$



$$\alpha\delta = \begin{cases} \textsc{Pos} & \text{if } \delta \in \{1, 2, \ldots\} \\ \textsc{Zero} & \text{if } \delta = 0 \\ \textsc{Neg} & \text{if } \delta \in \{-1, -2, \ldots\} \\ \bot_{\widehat{\mathbf{D}}} & \text{if } \delta = \bot_{\mathbf{D}} \\ \top_{\widehat{\mathbf{D}}} & \text{otherwise} \end{cases}$$

$$\iota\hat{\delta} = \begin{cases} \top_{\mathbf{D}} & \text{if } \hat{\delta} \in \{\top_{\widehat{\mathbf{D}}}, \textsc{Pos}, \textsc{Neg}\} \\ 0 & \text{if } \hat{\delta} = \textsc{Zero} \\ \bot_{\mathbf{D}} & \text{if } \hat{\delta} = \bot_{\widehat{\mathbf{D}}} \end{cases}$$

$$\begin{aligned} O_c &= \{\hat{+}, \hat{-}, \ldots\} \\ &\quad \text{where } \hat{+}\langle \textsc{Pos}, \textsc{Pos}\rangle = \textsc{Pos} \\ &\qquad\quad\;\; \hat{+}\langle \textsc{Pos}, \textsc{Neg}\rangle = \top_{\widehat{\mathbf{D}}} \\ &\qquad\qquad \vdots \end{aligned}$$

$$\begin{aligned} O_o &= \{\hat{>}, \ldots\} \\ &\quad \text{where } \hat{>}\langle \textsc{Pos}, \textsc{Neg}\rangle = [\![\text{true}]\!] \\ &\qquad\quad\;\; \hat{>}\langle \textsc{Neg}, \textsc{Neg}\rangle = \top_{\mathbf{D}} \\ &\qquad\qquad \vdots \end{aligned}$$

Figure 5: An example: excerpts from a sign `FACET_SPECIFICATION`.

1. A *concrete domain* **D** that must be lifted (as shown in Figure 3).
2. A *facet-value domain* **FacetValue** of finite height.
3. A monotonic *abstraction function* α : **D** $\rightarrow$ **FacetValue** and an associated monotonic *injection function* ι : **FacetValue** $\rightarrow$ **D** satisfying the safety condition that $id_{\mathbf{D}} \sqsubseteq \iota \circ \alpha$.
4. A set O of *abstract primitive operations*; an operation $\hat{p} \in O$ has functionality **FacetValue*** $\rightarrow$ **FacetValue**.

```
signature FACET =
sig
  structure Op: PRIM
  structure D: LIFTED
  eqtype FacetValue
  val join: FacetValue -> FacetValue -> FacetValue
  val abs: D.Lifted -> FacetValue
  val inj: FacetValue -> D.Lifted
  val O: Op.PRIM -> FacetValue list -> FacetValue
end
```

Figure 6: Components of a facet and the associated Standard ML signature.

5. A Partial-Evaluation Facet

Besides standard abstract interpretations such as the rule-of-signs abstraction, many useful semantic definitions can be captured in a facet specification. Particularly worthy of mention is the "partial-evaluation facet" proposed in [5]. One can create a facet to do constant folding by defining the abstract domain $\widehat{\mathbf{D}}$ to be the same as **D**. The abstraction and injection functions are then merely identity functions, and the abstract primitive operations, all closed, perform constant folding. In this way, the partial evaluation facet is itself exactly the semantic definition of the primitives of the language. It essentially encapsulates the primitive application clause of a simple partial evaluator (*e.g.*, the $\mathcal{K}_p$ function of Figure 2); hence, the name. However, it works just like any other facet and can be combined with other facets to integrate constant folding into the partial evaluation semantics.

6. Facets for Compound Data Types

Our first-order language has lists as a data type, but such compound types present difficult problems for abstract interpretations [14]. Given an abstraction, we would like to be able to "lift" that abstraction over lists. We have written a functor `ListSpec`, shown in Figure 9, for one possible way to provide this operation. It takes as arguments the abstract syntax and a facet specification defining some abstract interpretation. It returns a facet specification that is the result of "lifting" this abstract interpretation over lists. For example, `ListSpec(Absyn,SignSpec)` produces an abstract interpretation that maintains not only signs of integers, but lists of signs of integers, lists of lists of signs of integers, and so on. The code of `ListSpec` handles the list operations—`cons`,

$$
\begin{aligned}
\mathbf{D} &= \mathbf{D}_S \\
\textbf{FacetValue} &= \widehat{\mathbf{D}}_S \times \mathbf{D}_S \\
\alpha\delta &= \langle \alpha_S\delta, \delta \rangle \\
\iota\langle \hat{\delta}, \delta \rangle &= \delta
\end{aligned}
$$

$$
\begin{array}{lll}
\forall \hat{p}_S \in O_{cS}\; \exists \hat{p} \in O & \text{s.t.} & \hat{p}\langle \langle \hat{\delta}_1, \delta_1 \rangle, \ldots, \langle \hat{\delta}_n, \delta_n \rangle \rangle = \langle \hat{\delta}, \iota_S \hat{\delta} \rangle \\
 & & \text{where } \hat{\delta} = \hat{p}_S \langle \hat{\delta}_1, \ldots, \hat{\delta}_n \rangle \\
\forall \hat{p}_S \in O_{oS}\; \exists \hat{p} \in O & \text{s.t.} & \hat{p}\langle \langle \hat{\delta}_1, \delta_1 \rangle, \ldots, \langle \hat{\delta}_n, \delta_n \rangle \rangle = \langle \alpha_S \delta, \delta \rangle \\
 & & \text{where } \delta = \hat{p}_S \langle \hat{\delta}_1, \ldots, \hat{\delta}_n \rangle
\end{array}
$$

```
functor MakeFacet(structure S: FACET_SPECIFICATION): FACET =
struct
  structure Op = S.Op
  structure D = S.D
  type FacetValue = S.Dhat * D.Lifted
  fun join (dhat,d) (dhat',d') = (S.join dhat dhat', D.join d d')
  fun abs d = (S.abs d, d)
  fun inj (_,d) = d
  fun O po args =
      let val dhats = map (fn (dhat,_) => dhat) args
      in if (S.closedOp po) then
           let val dhat = (S.Oc po dhats)
           in (dhat, S.inj dhat)
           end
         else if (S.openOp po) then abs (S.Oo po dhats)
         else abs D.Top
      end
end
```

Figure 7: From facet specification S to facet.

$$
\begin{array}{rcl}
\mathbf{D} & = & \mathbf{D}_{F_1} (= \mathbf{D}_{F_2}) \\
\mathbf{FacetValue} & = & \mathbf{FacetValue}_{F_1} \times \mathbf{FacetValue}_{F_2} \\
\alpha \langle f_1, f_2 \rangle & = & \langle \alpha_{F_1} f_1, \alpha_{F_2} f_2 \rangle \\
\iota \delta & = & \iota_{F_1} \delta \sqcap \iota_{F_2} \delta \\
\hat{p} \in O \text{ is defined such that} & & \\
\hat{p} \langle \langle f_{1_1}, f_{2_1} \rangle, \ldots, \langle f_{1_n}, f_{2_n} \rangle \rangle & = & \begin{cases} f & \text{if } \iota f = \top_{\mathbf{D}} \\ \alpha(\iota f) & \text{otherwise} \end{cases} \\
& & \text{where } \; f \; = \; \langle \hat{p}_{F_1} \langle f_{1_1}, \ldots, f_{1_n} \rangle, \hat{p}_{F_2} \langle f_{2_1}, \ldots, f_{2_n} \rangle \rangle
\end{array}
$$

```
functor CombineFacets(structure F1: FACET
                      structure F2: FACET
                      sharing F1.Op = F2.Op
                      and F1.D = F2.D): FACET =
struct
  structure Op = F1.Op
  structure D = F1.D
  type FacetValue = F1.FacetValue * F2.FacetValue
  fun join (f1,f2) (f1',f2') = (F1.join f1 f1', F2.join f2 f2')
  fun abs d = (F1.abs d, F2.abs d)
  fun inj (f1,f2) = D.meet (F1.inj f1) (F2.inj f2)
  fun O po args =
      let val f = (F1.O po (map (fn (f1,_) => f1) args),
                   F2.O po (map (fn (_,f2) => f2) args))
      in case (inj f) of
           D.Top => f
         | => abs d
      end
end
```

Figure 8: From two facets F_1 and F_2 to one facet.

`car`, `cdr`, and `isnull`—and any other operation appropriate for the underlying abstraction (*e.g.*, `plus` or `gt` for `SignSpec`) is "passed down" to the underlying facet specification.

However, our `ListSpec` functor has a serious limitation. The definition of the abstract domain is

```
datatype Dhat = Top | Abs of S.Dhat | List of Dhat list | Bottom
```

where `S` is the underlying facet specification given as the argument. But since abstract lists are represented as SML lists, they cannot represent lists with an unknown ($\top$) tail. Essentially, applying `ListSpec` to a facet specification with abstract domain $\widehat{\mathbf{D}}$ produces a facet specification whose abstract domain is the separated sum $\sum_{i=1}^{\infty} \widehat{\mathbf{D}}^{i}$, which cannot represent a tail of $\top$ but is of finite height (given that $\widehat{\mathbf{D}}$ is of finite height).

One can imagine other approaches to handling lists. For instance, changing the `List` constructor of the above datatype to `Dhat*Dhat` would allow a richer specification of the tail of the list. But this of course would produce a domain of infinite height. An online partial-evaluator, since it subsumes evaluation and thus has no termination property, actually need not be constrained to operate only on finite-height domains. But, as we shall see in Section 8, we are interested in using facets for program analyses in general; hence our concern for finite-height representation of lists.

7. A Parameterized Online Partial Evaluator

We now have the machinery we need for an online partial evaluator that is parameterized with respect to any number of abstract interpretations. Figure 10 shows a semantics for a partial evaluator that is parameterized with respect to a facet F. The facet is not shown explicitly as an argument, but is assumed to be a "global variable" of the semantics. This parameterized semantics differs in several ways from the simple semantics shown in Figure 2:

- $\mathcal{E}$ now returns a facet value in addition to a residual expression. This allows us to keep abstract values, even for non-constant expressions, that might be useful in later computations.
- The input to the partial evaluator is now a list of facet values rather than a list of expressions. The facet thus gives us greater expressiveness with partial input, since we may input abstract values from the abstract domains that we have defined in the parameter facet's individual component facets.
- We no longer need the $\mathcal{K}_p$ function. All semantic definitions of primitive operations are in the facet and thus are parameterized.
- The $\mathcal{E}\,[\![c]\!]$ clause has changed; it now returns the abstraction of c as well as c itself.
- The $\mathcal{E}\,[\![p(e_1, \ldots, e_n)]\!]$ clause has changed. It no longer performs constant folding explicitely. Instead, it uses the abstract operators defined by the facet to perform the operation, and it uses the facet's injection function on the result to determine if any abstract interpretation yielded a constant. Constant folding can be integrated into this scheme with the partial-evaluation facet, as described in Section 5.

```
functor ListSpec(structure Absyn: ABSYN
                 structure S: FACET_SPECIFICATION
                 sharing Absyn.Op = S.Op
                 and type Absyn.Const = S.D.T): FACET_SPECIFICATION =
struct
  exception ListSpec
  structure Op = Absyn.Op
  structure D = S.D
  datatype Dhat = Top | Abs of S.Dhat | List of Dhat list | Bottom

  fun join d Bottom = d
    | join Bottom d = d
    | join (Abs sd) (Abs sd') = Abs (S.join sd sd')
    | join (List nil) (List nil) = List nil
    | join (List (d::l)) (List (d'::l')) =
      (case (join (List l) (List l')) of
           List l'' => List ((join d d') :: l'')
         | _ => Top)
    | join _ _ = Top

  fun abs D.Top = Top
    | abs (D.Exact (Absyn.List l)) = List (map (abs o D.Exact) l)
    | abs (c as D.Exact _) = Abs (S.abs c)
    | abs D.Bottom = Bottom

  fun inj Top = D.Top
    | inj (Abs sd) = S.inj sd
    | inj (List nil) = D.Exact (Absyn.List nil)
    | inj (List (d::l)) =
      (case ((inj d),(inj (List l))) of
           (D.Exact c, D.Exact (Absyn.List l)) => D.Exact (Absyn.List (c::l))
         | (D.Bottom,_) => D.Bottom
         | (_,D.Bottom) => D.Bottom
         | _ => D.Top)
    | inj Bottom = D.Bottom

  fun inject (Abs sd) = sd
    | inject d = S.abs (inj d)

  fun closedOp Op.CONS = true
    | closedOp Op.CAR = true
    | closedOp Op.CDR = true
    | closedOp po = S.closedOp po

  fun openOp Op.ISNULL = true
    | openOp po = S.openOp po

  fun Oc Op.CONS [Bottom,_] = Bottom       | Oc Op.CDR [Top] = Top
    | Oc Op.CONS [_,Top] = Top             | Oc Op.CDR [List (_::l)] = List l
    | Oc Op.CONS [d,List l] = List (d::l)  | Oc Op.CDR _ = Bottom
    | Oc Op.CONS _ = Bottom
                                           | Oc po args =
    | Oc Op.CAR [Top] = Top                  if (S.closedOp po)
    | Oc Op.CAR [List (d::_)] = d              then Abs (S.Oc po (map inject args)
    | Oc Op.CAR _ = Bottom                     else raise ListSpec

  fun Oo Op.ISNULL [Top] = D.Top
    | Oo Op.ISNULL [List nil] = D.Exact (Absyn.Tr true)
    | Oo Op.ISNULL [List _] = D.Exact (Absyn.Tr false)
    | Oo Op.ISNULL _ = D.Bottom

    | Oo po args =
      if (S.openOp po) then S.Oo po (map inject args)
      else raise ListSpec
end
```

Figure 9: Making facet specifications for lists.

- The $\mathcal{E}$ $[\![\text{if } e_1\, e_2\, e_3]\!]$ clause has changed in the case where the branch cannot be selected and a residual $[\![\text{if}\ldots]\!]$ expression is constructed. In that case, the join of the facet values of the partial evaluations of e_1 and e_2 is returned as the facet value of the result. In this way, we keep the most accurate information consistent with both branches.

Figure 11 shows the signature `ONLINE` and most of the code of the `Online` functor that implements the partial evaluator defined in Figure 10. The functor takes as a parameter a facet whose concrete domain must be (via a sharing constraint) $\widehat{\mathbf{Const}}$, the lifted domain of constants of the language, as shown in Figure 3.

The functor returns a structure that matches the `ONLINE` signature. This structure contains a function `online` which maps a program and a list of facet values to a residual program. The `int` argument to `online` specifies the maximum depth of conditional branching that may precede an unfolding of a function call. The list of facet values provide partial input to the program. The facet parameter defines the expressiveness of this partial input; by adding more facets with `CombineFacets`, one can increase power and expressiveness.

Figure 12 shows some actual runs of the online evaluator. The evaluator was given a facet formed from the following facet specifications:

`SignSpec`	The sign abstraction of integers shown in Figure 5.
`PESpec`	The constant-folding facet described in Section 5.
`LengthSpec`	A length abstraction of lists.
`ListSpec(Absyn,SignSpec)`	Signs mapped over lists.
`ListSpec(Absyn,PESpec)`	Constants mapped over lists.

8. Other Examples and Further Work

Binding-time analysis can be parameterized in a similar fashion. As mentioned in Section 2, [5] presents the idea of *abstract facets* with values STATIC and DYNAMIC and a *facet analysis* to map facets to abstract facets. We instead merely add a binding-time facet to the facets that we already have defined. A binding-time facet is one that implements the standard binding-time abstraction. I.e., the abstract domain is $\bot \sqsubseteq \text{STATIC} \sqsubseteq \text{DYNAMIC}$, and $\hat{p}\langle \hat{d}_1, \ldots, \hat{d}_n \rangle = \text{STATIC}$ iff $\forall(1 \le i \le n)$ $\hat{d}_i = \text{STATIC}$. The advantage here is that we can combine this binding-time facet with the facets we already used for the online partial evaluator, resulting in a potentially more powerful domain for the binding-time analysis. This advantage is illustrated in the last `factiter` and the `factfunny` examples of Figure 13. In the `factfunny` example, the partial evaluation facet is allowed to actually do the constant folding to produce 0 for the `minus` operation. Then the sign facet, after picking up the 0, returns 0 for the operation `times(`$\top$`,`ZERO`)`. The result is that the binding-time analysis returns STATIC for the `r` parameter.

Future work includes:

- Extension to a CPS-based language, such as the intermediate language of the SML of New Jersey compiler.

- Syntactic Domains
  (Same as in Figure 2.)
- Semantic Domains

$$\begin{array}{rcll} f & \in & \textbf{FacetValue} & = \textbf{FacetValue}_F \\ r & \in & \textbf{Residual} & = \textbf{FacetValue} \times \textbf{Exp} \\ \rho & \in & \textbf{Env} & = \textbf{Var} \rightarrow \textbf{Residual} \end{array}$$

- Valuation Functions

$$\begin{array}{rcl} \mathcal{P} & : & \textbf{Prog} \rightarrow \textbf{FacetValue}^* \rightarrow \textbf{Int} \rightarrow \textbf{Prog} \\ \mathcal{E} & : & \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{Prog} \rightarrow \textbf{Residual} \times \textbf{Prog} \\ \text{APP} & : & \textbf{Fn} \rightarrow \textbf{Residual}^* \rightarrow \textbf{Env} \rightarrow \textbf{Program} \rightarrow \textbf{Residual} \times \textbf{Program} \end{array}$$

$$\begin{array}{rcl}
\mathcal{P}\,\sigma\,\langle f_1, \ldots, f_{n_1}\rangle & = & (\mathcal{E}[\![f_1(x_1, \ldots, x_{n_1})]\!](\bot[f_k/x_k])\sigma) \downarrow_2 \\
& & \text{where } \sigma = [\![\{f_i(x_1, \ldots, x_{n_i}) = e_i\}]\!] \\
\\
\mathcal{E}\,[\![c]\!]\,\rho\,\sigma & = & \langle\langle \alpha_F c, [\![c]\!]\rangle, \sigma\rangle \\
\mathcal{E}\,[\![x]\!]\,\rho\,\sigma & = & \langle \rho[\![x]\!], \sigma\rangle \\
\mathcal{E}\,[\![p(e_1, \ldots, e_n)]\!]\,\rho\,\sigma & = & (\iota_F f \in \textbf{Const}) \rightarrow \langle\langle f, \iota_F f\rangle, \sigma_n\rangle,\ \langle\langle f, [\![p(e'_1, \ldots, e'_n)]\!]\rangle, \sigma_n\rangle \\
& & \text{where } \begin{array}[t]{rcl} \langle\langle f_1, e'_1\rangle, \sigma_1\rangle & = & \mathcal{E}[\![e_1]\!]\rho\sigma \\ & \vdots & \\ \langle\langle f_n, e'_n\rangle, \sigma_n\rangle & = & \mathcal{E}[\![e_n]\!]\rho\sigma_{n-1} \\ f & = & \hat{p}_F\langle f_1, \ldots, f_n\rangle \qquad (\hat{p}_F \in O_F) \end{array} \\
\mathcal{E}\,[\![\text{if } e_1\, e_2\, e_3]\!]\,\rho\,\sigma & = & (e'_1 \in \textbf{Const}) \rightarrow \\
& & \quad (e'_1 \rightarrow \mathcal{E}[\![e_2]\!]\rho\sigma_1,\ \mathcal{E}[\![e_3]\!]\rho\sigma_1), \\
& & \quad \langle\langle f_2 \sqcup f_3, [\![\text{if } e'_1\, e'_2\, e'_3]\!]\rangle, \sigma_3\rangle \\
& & \quad \text{where } \begin{array}[t]{rcl} \langle\langle f_2, e'_2\rangle, \sigma_2\rangle & = & \mathcal{E}[\![e_2]\!]\rho\sigma_1 \\ \langle\langle f_3, e'_3\rangle, \sigma_3\rangle & = & \mathcal{E}[\![e_3]\!]\rho\sigma_2 \end{array} \\
& & \text{where } \langle\langle f_1, e'_1\rangle, \sigma_1\rangle \ = \ \mathcal{E}[\![e_1]\!]\rho\sigma \\
\mathcal{E}\,[\![f(e_1, \ldots, e_n)]\!]\,\rho\,\sigma & = & \text{APP}[\![f]\!]\langle r_1, \ldots, r_n\rangle\rho\sigma_n \\
& & \text{where } \begin{array}[t]{rcl} \langle r_1, \sigma_1\rangle & = & \mathcal{E}[\![e_1]\!]\rho\sigma \\ & \vdots & \\ \langle r_n, \sigma_n\rangle & = & \mathcal{E}[\![e_n]\!]\rho\sigma_{n-1} \end{array}
\end{array}$$

Figure 10: A parameterized online partial evaluator for a strict first-order functional language.

```
signature ONLINE = sig
  structure Absyn: ABSYN
  structure F: FACET
  val online: int -> Absyn.Prog -> F.FacetValue list -> Absyn.Prog
end

functor Online(structure Absyn: ABSYN
               structure F: FACET
               sharing Absyn.Op = F.Op
               and type Absyn.Const = F.D.T): ONLINE = struct
  structure Absyn = Absyn
  structure F = F

  fun getf (f,_) = f              fun getexp (_,exp) = exp

  fun P (Absyn.Program prog) input ud = ...

  and E (exp as (Absyn.Constexp c)) _ prog _ =
      ((F.abs (F.D.Exact c), exp), prog)

    | E (Absyn.Varexp var) env prog _ = (env var, prog)

    | E (Absyn.Pexp (po,explst)) env prog ud =
      let val (residlst,prog') = Elist explst env prog ud
          val f = F.O po (map getf residlst)
      in case (F.inj f) of
           F.D.Exact c => ((f, Absyn.Constexp c), prog')
         | _ => ((f, Absyn.Pexp (po, (map getexp residlst))), prog')
      end

    | E (Absyn.Fexp (name,explst)) env prog ud =
      let val (residlst,prog') = Elist explst env prog ud
      in APP name residlst env prog' ud
      end

    | E (Absyn.Ifexp (exp1,exp2,exp3)) env prog ud =
      let val ((_,e1),prog1) = E exp1 env prog ud
      in case e1 of
           Absyn.Constexp (Absyn.Tr true) => E exp2 env prog1 ud
         | Absyn.Constexp (Absyn.Tr false) => E exp3 env prog1 ud
         | _ => let val ((f2,e2),prog2) = E exp2 env prog1 (ud-1)
                    val ((f3,e3),prog3) = E exp3 env prog2 (ud-1)
                in ((F.join f2 f3, Absyn.Ifexp (e1,e2,e3)), prog3)
                end
      end

  and Elist (exp::rest) env prog ud =
      let val (resid,prog') = E exp env prog ud
      in let val (rest',prog'') = Elist rest env prog' ud
         in ((resid::rest'),prog'')
         end
      end
    | Elist _ _ prog _ = (nil,prog)

  and APP name residlst env prog ud = ...
      (* decides whether or not to specialize based on the unwind depth *)
  fun online ud (program as (Absyn.Program _)) input = ...
      (* adds an expression to each facet value in input and calls P *)
end
```

Figure 11: The parameterized online partial evaluator.

Source program:

```
sort(lst) =
  if isnull(lst)
    []
    insert(car(lst),
           sort(cdr(lst)))

insert(n,lst) =
  if isnull(lst)
    cons(n,[])
    if gt(n,car(lst))
      cons(car(lst),
           insert(n,cdr(lst)))
      cons(n,lst)
```

Input `[top,2,1]` yields the residual program:

```
sort1(lst) = insert8(car(lst))
insert8(n) = if gt(n,1)
             cons(1,if gt(n,2)
                 cons(2,cons(n,[]))
                 cons(n,[2]))
             cons(n,[1,2])
```

Input `[neg,3,1,2]` yields the residual program:

```
sort12(lst) = cons(car(lst),[1,2,3])
```

Input `[pos,neg]` yields the residual program:

```
sort7(lst) = cons(car(cons(car(car(lst)),
                            [])),
                  cons(car(lst),[]))
```

Figure 12: Example runs of the online evaluator.

```
factiter(n,r) =
  if eq(n,0)
    r
    factiter(minus(n,1),
             times(n,r))
```

input	output
[3,1]	[static,static]
[3,top]	[static,top]
[top,1]	[top,top]
[top,0]	[top,0]

```
factfunny(n,i,j,r) =
  if eq(n,0)
    1
    factfunny(minus(n,1),i,j,
              times(n,times(r,minus(i,j))))
```

input	output
[top,3,3,static]	[top,3,3,static]

Figure 13: Examples of the binding-time analyzer.

- Other collecting interpretations (*e.g.*, strictness analysis). The success of the parameterized binding-time analyzer lends promise to this line of research.
- A specializer to work with the binding-time analyzer.
- Development of mathematical foundations to account for the differences from the original framework of [5].

9. Conclusions

The notion of facet parameterization provides a nice way to combine abstract interpretation with partial evaluation. We have used the modules system of Standard ML to develop an implementation that makes facet specification easy, and we have successfully implemented an online partial evaluator based on this scheme. Furthermore, we have implemented a binding-time analyzer that can use the exact same facet parameter. This indicates that this specification of facets may be useful for collecting interpretations in general.

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, England, 1987.

[2] A. Bondorf and O. Danvy. *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types.* Technical Report 90–4, DIKU, University of Copenhagen, Denmark, 1990.

[3] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger (Ed.), *Proceedings of the Second European Symposium on Programming '88*, Nancy, France (March 1988), *Lecture Notes in Computer Science*, Vol. 300, Springer-Verlag, 1988, 236–246.

[4] C. Consel. Binding time analysis for higher order untyped functional languages. *Proceedings of the 1990 Conference on Lisp and Functional Programming*, Nice, France, June 1990, 264–272.

[5] C. Consel and S. C. Khoo. Parameterized partial evaluation. *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, 92–106.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977, 238–252.

[7] Jesper Jørgensen. Generating a pattern matching compiler by partial evaluation. To appear at POPL'92.

[8] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus. *ACM Transactions on Programming Languages and Systems*, To appear.

[9] P. Hudak and J. Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 2, April 1991, 269–290.

[10] N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In Jean-Pierre Jouannaud (Ed.), *Rewriting Techniques and Applications*, Dijon, France, *Lecture Notes in Computer Science*, Vol. 202, Springer-Verlag, 1985, 124–140.

[11] N. Jones, P. Sestoft, and H. Søndergaard. MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, Vol. 2, No. 1, 1989, 9–50.

[12] D. B. MacQueen. Modules for Standard ML. In R. Harper, D. MacQueen, and R. Milner, *Standard ML*, Technical Report ECS–LFCS–86–2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[13] Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones (Eds.), *Partial Evaluation and Mixed Computation*, North-Holland, 1988, 325–347.

[14] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). *Abstract Interpretation of Declarative Languages*. Abramsky and Hankin (Eds.), Ellis Horwood, Chichester, England, 1987, 266-275.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.