ML Partial Evaluation using Set-Based Analysis

Karoline Malmkjær  Nevin Heintze  Olivier Danvy

February 1994
CMU-CS-94-129
ML Partial Evaluation using Set-Based Analysis

Karoline Malmkjær  Nevin Heintze  Olivier Danvy

_February 1994_
CMU-CS-94-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

(Also appears as Fox Memorandum CMU-CS-FOX-94-04.)

This work was partially sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title “The Fox Project: Advanced Development of Systems Software”, ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168. It was also partially sponsored by the Danish Research Councils through the DART project.

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 U.S. Government.
Keywords: partial evaluation, ML, set-based analysis, set constraints, binding
time analysis, program specialization, continuation-based specialization.
Abstract

We describe the design and implementation of an off-line partial evaluator for Standard ML programs. Our partial evaluator consists of two phases: analysis and specialization.

Analysis: Set-based analysis is used to compute control flow, data flow and binding-time information. It provides a combination of speed and accuracy that is well suited to partial-evaluation applications: the analysis proceeds at a few hundred lines per second and is able to deal with higher-order functions, partially-static values, arithmetic, side effects and control effects.

Specialization: To treat the rich static information supplied by set-based analysis, continuation-based specialization is used in conjunction with a notion of "lightweight" symbolic values. The specializer adapts and improves upon the proven design principles of off-line polyvariant partial evaluators.

Our system is integrated into the New Jersey compiler for Standard ML: both input and output languages are the compiler's intermediate language LAMBDA. As such, our ML partial evaluator is not a source-ML to source-ML program transformer and issues of desugaring and type checking are avoided.

The core part of our implementation, handling higher-order programs with partially-static values, is complete. We are currently working on extensions of the specializer to treat computational effects.
I Introduction

The last decade has seen substantial advances in partial evaluation [7, 12, 13]. In particular, it now appears feasible to build practical large-scale partial evaluators. At the same time, much effort has been directed towards building well-structured systems software. Such efforts provide a rich source of motivating examples for partial evaluation.

The specific context of our work is the CMU FOX project, which addresses modular systems building in ML. Core parts of the standard TCP/IP network protocol suite have been implemented in an exceptionally structured and modular way [2]. However, the extra modularization introduces additional run-time overhead. Partial evaluation provides an appropriate tool for the removal of this overhead.

We aim to construct a system that provides practical and effective partial evaluation of systems software written in ML. Such a system must address all aspects of ML, including side-effects, control and arithmetic.

Our design uses “polyvariant specialization”, which is an aggressive forward constant propagation tightly paired with the multiple specialization of selected source program points and driven by a global, off-line “binding-time” analysis. Specifically:

**Binding-time analysis**: The program is pre-processed to compute information about the static structure of values available at partial-evaluation time [5, 12, 16].

**Specialization**: The program is processed to propagate static values, perform static computation and construct the residual program.

The technique is simple and effective.

In contemporary off-line partial evaluators [12], the binding-time analyzer is designed to meet the needs of the specializer. In contrast, we do not have a tailored binding-time analyzer. Instead, we obtain binding-time information by enriching an independent and more generic source: set-based analysis [9]. Correspondingly, our specializer is designed not only to use the binding-time information but also to exploit set-based information.

Our system is integrated into the New Jersey ML compiler [1]. The compiler’s LAMBDA intermediate language is used for both the input and output of the evaluator. As such, our ML partial evaluator is not a source-ML to source-ML program transformer. Instead, it operates on type-checked and syntactically-desugared programs. This approach differs substantially from that of Mix-style partial evaluators such as Similix and (to a lesser extent) Schism [4, 6]. Our system also differs from
Birkedal and Welinder's recent SML-Mix, which is a partial-evaluation compiler tailored to generate partial evaluators dedicated to source ML programs [3].

ML has a powerful static semantics. For this reason we have chosen to partially evaluate programs after the static semantics is processed. In this way the partial evaluator can focus on the main goal of partial evaluation — removal of run-time overhead — instead of spending resources on irrelevant issues such as processing of ML static semantics.

One could speculate about the properties our partial evaluator should satisfy — besides the definitional one, i.e., running the specialized program on the remaining input must yield the same result as running the source program on the complete input (modulo termination). For example, should our partial evaluator preserve ML typing? However, the question is not very meaningful for two reasons. First, LAMBDA is not a typed language. Second, the compiler uses transformations that do not preserve ML typability, e.g., CPS transformation [8] and also lambda-lifting.

2 System Structure

Before describing the analysis and specialization components of our system in detail (Sections 3 and 4 respectively), we give an overview of our system:

1. The input ML program is first parsed, type-checked, and translated from abstract syntax into LAMBDA code (this is all performed by the front-end of the New Jersey compiler).

2. The LAMBDA code is then pre-processed. Intermediate results are named and some trivial code simplifications are performed. The names are used in the communication between the analyzer and the specializer.

3. The analysis phase of the partial evaluator is applied to the LAMBDA code output by the pre-processing stage. The analyzer constructs a mapping from variable identifiers into control-flow, data-flow and binding-time information.

4. The mapping from the analysis phase and the LAMBDA code from the pre-processing phase are then passed to the specializer. The output of the specializer is another LAMBDA program.

5. The output of the specializer is compiled by the back-end of the New Jersey compiler, and made available via a variable binding in the top-level environment.
The choice of LAMBDA to represent programs throughout deserves comment. LAMBDA provides a simple representation of programs (there are only a small number of different kinds of expressions), and we avoid many of the issues of syntactic processing and type checking that arise in the manipulation of ML source programs. However, LAMBDA is not ideally suited for analysis and program manipulation for a number of reasons. The pre-processing (stage 2) addresses some of these problems. Ideally, one would like to have properties such as “each intermediate result is named” and “operations are only applied to trivial expressions” to be built into the definition of the representation. Sequentialization of intermediate computations would also simplify program manipulation. (NB: These properties are all met in nqCPS, A-normal forms, monadic normal forms, and higher-order three-address code.)

3 Analysis

The set-based approach to program analysis employs a single notion of approximation: all inter-variable dependencies are ignored [9, 10, 11]. This is achieved by treating variables as sets of values. In other words, the environments encountered at each point in a program are collapsed into a single set environment (mapping variables into sets). In effect, analysis is carried out by extracting relationships between the sets of values for the program variables, and then reasoning about these relationships. For example, when set-based analysis is applied to the program:

```plaintext
let fun append(nil, y') = y'
    | append(x :: xs, y) = x :: append(xs, y)
fun rev nil = nil
    | rev(z :: zs) = append(rev zs, [z])
in rev [1,2,3,4]
end
```

the result of the program is approximated by the set of all lists constructed from 1, 2, 3 and 4. In effect, the shape of the list is preserved, but information about the order and length of the list are lost. A key difference from abstract-interpretation approaches to program analysis is that set-based analysis does not use an underlying abstract domain to approximate program values.

To obtain binding-time information, set-based analysis must be extended to reason about unknown values or parameters. These parameters are manipulated and propagated by the analysis so that the output of the analysis is correct for all instantiations of the parameter. That is, set-based analysis must take into account all possible behaviors of the parameter. For example, in the context of the definition
of append, suppose that we call append([1,2], dynamic), where the token
dynamic is a parameter indicating a dynamic value. Set-based analysis yields the
following description:

\[
\begin{align*}
X &= 1 \cup 2 \\
L &= \text{dynamic} \cup (X : \llbracket L \rrbracket)
\end{align*}
\]

where \( \text{L} \) is the set variable describing the results of the call to append. The next
element illustrates a combination of polyvariant analysis and parametric reasoning:

```haskell
fun map f nil nil = nil |
map f (x :: l) = (f x) :: (map f l)
val t = [1,2,3]
val d = dynamic
val u = map (fn x => (x, d)) t
val v = map (fn (x, y) => x) u
val w = map (fn (x, y) => y) u
```

For this program, set-based analysis yields the following information about the
variables \( u, v \) and \( w \): \( u \) is a list of pairs whose first element is either 1, 2 or 3 and
whose second argument is dynamic; \( v \) is a list of 1's, 2's and 3's, and \( w \) is a list of
dynamic's.

We conclude this brief overview by illustrating some important issues arising
in set-based binding-time analysis. Consider the following program fragment:

```haskell
val (v1, v2) = dynamic
val w = if dynamic then 1 else 2
```

In the first statement, \( v_1 \) and \( v_2 \) de-construct the parameter dynamic. This is
modeled by introducing a derived parameter \( \text{subterm} \text{(dynamic)} \), whose purpose
is to denote the set of all subterms of the parameter dynamic. In the second
statement, a standard set-based analysis would just compute the set \( \{1,2\} \) for \( w \);
however it is important to propagate the fact that the value of \( w \) is not statically
determined. When a test is dependent on a dynamic parameter, we therefore
introduce a \( \text{computation} \text{(dynamic)} \) parameter to the result of the if statement, so
that the eventual set computed for \( w \) is \( \{1,2\} \cup \text{computation} \text{(dynamic)} \).

4 Specialization

We use off-line, polyvariant specialization. It is off-line because m : t of the
control decisions needed in the specializer are determined by information from the
analyzer. It is polyvariant because it selects a set of specialization points in the
source program and produces a residual program consisting of multiple variants
of these. Furthermore, we handle higher-order and partially-static values. Since
we want a flexible, extensible, and efficient system, we have aimed for a simple
design, re-using concepts that have proven successful in previous off-line partial
evaluators.

Since we operate in LAMBDA, we avoid many problems often associated with
partial evaluation for typed languages. On the other hand, it also imposes some
restrictions. It is for example crucial to be able to process partially-static values
since in LAMBDA, function parameters are passed as records.

The specializer is guided by the information computed by the analyzer. Using
the static input, the specializer starts at the entry point of the source program and
propagates constant values through the program by unfolding function calls and
reducing static expressions. Residual code is generated whenever parts of the
source program cannot be statically evaluated.

Some source functions are recursive, and often there is not enough information
to execute their calls statically. Instead we need to construct residual recursive
functions. A simple mechanism is to select specialization points in the source pro-
gram. When the specializer meets a specialization point, it generates residualized
code representing a specialized version of the specialization point. (This is referred
to as either "polyvariant specialization", "customization", or "procedure cloning"
in the literature.)

As specialization points, we use conditional expressions whose test is dynamic.
Experience suggests that this is a good choice [12], but further optimization is
possible [15]. In a pre-pass through the program, the specialization points are
lambda-lifted into a collection of global recursive equations. The free variables of
each conditional expression are given as parameters to the recursive equation. We
remark that whereas function calls are unfolded, calls to global recursive equations
(the specialization points) are always residualized. The residual program is thus a
collection of recursive equations.

Effective unfolding requires finer descriptions of values than given by the
course "known / unknown" distinction. Such descriptions include higher-order
and partially static values. Partial evaluators typically represent these symboli-
cally. This representation causes problems in the presence of call unfolding,
such as computation duplication, reordering, and loss of dynamic computations.
Our design solves this by naming residual computations. However, naming has
the disadvantage of limiting the static data flow. This shortcoming can be ad-
dressed by using continuation-based specialization. In effect, continuations are used to communicate across naming: the continuation accounts for the specialization context, and is sent the name of the residual expression. More detail about continuation-based specialization can be found elsewhere [12, Sec. 10.5] [14].

Since residual computations are named, any dynamic subpart of a symbolic value is a variable. To simplify the treatment of symbolic values at specialization points, we represent them as pairs: the actual symbolic value (simple, higher order, or partially static) and the list of its free dynamic variables. We call this representation lightweight symbolic values.

We illustrate elements of our design with the following declaration of and call to a specialization point:

```latex
let fun foo (x, y, z) = ... 
    in ... foo (a, b, c) ... 
    end
```

Suppose that when the call `foo (a, b, c)` is processed, the first argument `a` is the partially-static value `(CON1 (vl, 10), [vi])`: the construction `CON1` of a dynamic variable `vl`, and the number 10, paired with the list of its free variables. The second argument is the higher-order value `(fn x => x + y, (y = (CON2 (v2,20), [v2]), [v2]))`: a closure with one free variable `y` bound to another partially-static value. The third argument is a constant string ("hello world", [])

Then we obtain the following residual specialization point and corresponding residual call:

```latex
let fun foo' (vl', v2') = ... 
    in ... foo' (vl, v2) ... 
    end
```

where `foo', vl', and v2'` are fresh names. All the static information about the arguments of `foo` has been propagated into the body of `foo'`. All the dynamic information has been filtered out and residualized in the call to `foo'` — without traversing any symbolic values. Note that, in general, there is no relationship between the number of arguments of the specialization point and the number of arguments of its residualized versions.

The specializer is designed to be simple, efficient, and extensible. Set-based analysis itself provides one possible extension: it yields more information than
a typical binding-time analysis. Instead of stating that an identifier denotes a (partially) static or dynamic value, it describes a set of values for the identifier. This provides an opportunity for much finer program specialization. Set-based analysis also provides information about side effects and control effects. We are currently extending the specializer to exploit that information.

5 Conclusion

We are building a partial evaluator for Standard ML programs. Our partial evaluator is implemented as an extension of the New Jersey compiler (and is itself written in ML). It is structured as an off-line system consisting of a set-based binding-time analysis and a continuation-based specializer.

The goals of our design are simplicity, robustness, and efficiency. It streamlines a number of concepts from previous partial evaluators. Moreover, lightweight symbolic values provide an efficient representation of static values.

One of the original motivations for this work is the FOX project [2]. This project uses ML for building system software, and crucially needs a powerful compile-time optimizer. For various reasons, existing implementations of ML do not use global flow analyses or partial evaluation. Our work is one of the first efforts to employ global analysis for ML program transformation and optimization.

References


Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment or administration of its programs on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state or local laws, or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state or local laws, or executive orders.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.