Partially Static Operations

Peter Thiemann
University of Freiburg, Germany
thiemann@acm.org

Abstract

Partial evaluation distinguishes between different binding times when manipulating values in a program. A partial evaluator performs evaluation steps on values with a static binding time whereas it generates code for values with a dynamic binding time. Binding time descriptions have evolved from monolithic to fine grained, partially static data structures where different components may have different binding times.

We consider the next step in this evolution where not just data constructors but also operations can give rise to partially static structures, which are processed by the partial evaluator. We introduce this notion in an online setting, generalize it, generate a binding-time analysis for it, and implement an offline program generator supporting partially static operations with it. We report on our initial experiences with this new approach.

General Terms Experimentation, Languages, Theory

Keywords partial evaluation, algebraic simplification

1. Introduction

A program specializer transforms a program by taking advantange of known properties of the input data. The goal is to improve on some non-functional aspect of the program, such as efficiency, while keeping the semantics unchanged. Partial evaluation restricts the range of the properties to being equal to a particular value. The degree of knowledge about a value is called its *binding time*.

In their seminal work on self-applicable partial evaluation, Neil Jones, Peter Sestoft, and Harald Søndergaard [1] devise a program analysis that assigns one of two monolithic binding times to each subexpression of a program: "Static" means that a value is fully available at specialization time, whereas "dynamic" classifies a value that may not be available. They had to repeatedly rewrite their specializer to amend for the weaknesses of this coarse analysis. Subsequent work has strengthened the power of the specializer and increased the expressiveness of the binding-time analysis to obtain satisfactory specialization for a wider range of programs. In particular, state-of-the art specializers are able to process partially static data (like lists of known length, but with unknown elements) [4] and contain corresponding binding-time analyses.

Our approach generalizes the idea of partially static data in the quest to improve the specialization of numeric code. As an example, let a, b, and d be static numbers and let c be a dynamic number and apply a partial evaluator to the expression

$$(a + b) + (c + d).$$

Binding-time analysis annotates this expression as follows:

$$\lceil (a +^S b) \rceil +^D (c +^D \lceil d \rceil)$$

This annotation instructs the partial evaluator to add a and b at specialization time. It *lifts* the result and d to run time and generates code for the remaining two additions. Lifting $\lceil \cdot \rceil$ is the function that maps a number from its run-time representation to its source representation.

However, this approach is overly conservative. If the specializer were aware of the commutativity and associativity of addition, then it could reorder the expression before specialization:

$$((a + b) + d) + c$$

This equivalent expression has better binding-time properties. Two additions can be executed at specialization time and just one addition is deferred to run time.

$$\lceil ((a +^S b) +^S d) \rceil +^D c$$

This paper examines ways of incorporating laws like commutativity and associativity in a partial evaluation algorithm without loosing its compositionality. The main idea is to have the specializer manipulate symbolic, partially static representations of values that expose enough of its structure to enable additional static operations and decrease the number of dynamic operations.

2. Specialization with Partially Static Operations

The specialization algorithm for partially static operations represents values in the form of *admissible patterns* P, which is a set of arithmetic terms that includes at least all numeric literals and variables (that is, $P_{std} := \mathbb{F} \cup \mathsf{Var} \subseteq P$). Each operation is first applied symbolically. Then the specializer applies laws and evaluation rules to simplify the resulting term with the goal of fitting it into the set P. In doing so, the specializer may have to abstract subterms to variables, which leads to generating code for these subterms. Law applications and subterm abstractions are chosen so as to minimize the number of operations in the generated code.

Simplification is a function that decides what to transform into a pattern and what to generate code for. It transforms a term into an equivalent output pattern where zero or more definitions have been abstracted to make the result term fit into a pattern. The definitions correspond to generated code whereas the pattern is the structure that is further propagated by the specializer. The generated code may bind variables used in the output pattern. Using Term for the set of arithmetic terms and letting $\mathsf{Def} = \mathsf{Var} \times \mathsf{Term}$ be the set of variable definitions, we can specify the simplification function:

$$\mathcal{SI} : \mathsf{Term} \to \mathsf{Def}^* \times P$$
$$\mathcal{SI}(e) = ([x_1 = e_1, \ldots, x_k = e_k], p)$$

PEPM'13, January 21–22, 2013, Rome, Italy.
ACM 978-1-4503-1842-6/13/01.

such that $\mathcal{E} \vdash e \longleftrightarrow^* p[x_i \mapsto e_i]$ where $\mathcal{E}$ is the set of rewrite rules that describes the applicable laws. $\mathcal{SI}$ chooses some $p[x_i \mapsto e_i]$ such that the combined cost of $e_1, \ldots, e_k$ is minimal.

Now we can describe the specialization of an operation. Let Ops_n be the set of operation symbols of arity n. The function $\mathcal{OP}_n$ applies an n-ary operator to n patterns and simplifies the result.

$$\mathcal{OP} : (n : \mathbb{N}) \to \mathsf{Ops}_n \to P^n \to \mathsf{Def}^* \times P$$
$$\mathcal{OP}_n(f)(p_1, \ldots, p_n) = \mathcal{SI}(f(p_1, \ldots, p_n))$$

The minimality criterion of $\mathcal{SI}$ rules out trivial solutions. Consider the specializer for multiplication with pattern set P_{std}.

$$\begin{array}{llll} \mathcal{OP}_2(*)(\underline{x}, 1) & \neq & ([z = x * 1], \underline{z}) & \text{violates minimality} \\ \mathcal{OP}_2(*)(\underline{x}, 1) & = & ([\,], \underline{x}) & \text{ok} \\ \mathcal{OP}_2(*)(\underline{x}, \underline{y}) & = & ([z = x * y], \underline{z}) & \text{ok} \end{array}$$

This setting is directly applicable to specializing exponentiation.

```
power (x, n) =
  if n==0 then 1 else x * power (x, n-1)
```

If `x` is dynamic and `n` is static, then the operations `==` and `-` are always evaluated and do not lead to code generation.

Specializing `power` in this context demonstrates that trivial multiplications with `1` are elided (a similar effect has been achieved using a hand-written staged program[2]):

```
power_0 (x) = 1
power_1 (x) = x
power_2 (x) = let z = x*x in z
power_3 (x) = let z1 = x*x in let z2 = x*z1 in z2
```

3. Binding-Time Analysis

Binding-time analysis has been developed to achieve efficient self application [1]. Later it has become instrumental in the direct derivation of code generators from binding-time annotated programs (the cogen approach) [3]. Our work pursues the latter goal.

The simplification operation allows us to derive a binding-time analysis based on abstract interpretation of the pattern-driven specialization. The idea for the analysis domain is to abstract over values that are static, but unknown at analysis time. To this end, we abstract the specialization-time values (patterns) to *abstract patterns* $\widehat{P}$ determined by the following grammar:

$$\begin{array}{lll} \widehat{P}_0 & ::= & \underline{X} \mid \mathbb{F} \mid \top \mid f^{(n)}(\widehat{P}_0^n) \\ \widehat{P} & ::= & \bot \mid \widehat{P}_0 \mid \widehat{P} \cup \widehat{P} \end{array}$$

Each abstract pattern determines a set of terms via the concretization function $\gamma : \widehat{P} \to 2^{\mathsf{Term}}$.

$$\begin{array}{llll} \gamma(\bot) & = & \emptyset & \gamma(\widehat{p}_1 \cup \widehat{p}_2) = \gamma(\widehat{p}_1) \cup \gamma(\widehat{p}_2) \\ \gamma(\underline{X}) & = & \mathsf{Var} & \\ \gamma(a \in \mathbb{F}) & = & a & \gamma(f^{(n)}(\widehat{p}_1, \ldots \widehat{p}_n)) = \\ \gamma(\top) & = & \mathbb{F} & \{f(p_1, \ldots, p_n) \mid p_i \in \gamma(\widehat{p}_i)\} \end{array}$$

The abstraction of the operations is entirely standard. Let α be the uniquely determined abstraction function that completes γ to a Galois connection between concrete and abstract patterns.

$$\widehat{\mathcal{OP}} : (n : \mathbb{N}) \to \mathsf{Ops}_n \to \widehat{P}^n \to \widehat{P}$$
$$\widehat{\mathcal{OP}}_n(f)(\widehat{p}_1, \ldots, \widehat{p}_n) = \bigsqcup\{\alpha(\#2(\mathcal{OP}_n(f)(p_1, \ldots, p_n))) \mid p_i \in \gamma(\hat{p}_i)\}$$

The function $\#2$ projects the second component of a pair.

For example, $\alpha(P_{std}) = \top \cup \underline{X}$ and then abstracting the multiplication operation from the previous examples yields:

$$\widehat{\mathcal{OP}}_2(*)(\underline{X}, 1) = \underline{X}$$
$$\widehat{\mathcal{OP}}_2(*)(\underline{X}, \underline{X}) = \underline{X}$$
$$\widehat{\mathcal{OP}}_2(*)(\underline{X}, 1 \cup \underline{X}) = \underline{X}$$

An abstract interpretation over the domain of abstract patterns annotates each subexpression with a union of possible patterns. This analysis serves as a binding-time analysis for the specializer with partially static operations. The table below shows an excerpt of the results of the corresponding fixpoint computation on the `power` example. The abstractions for `1`, `x`, and `n` are, respectively, 1, $\underline{X}$, and $\top$, which indicates that the analysis distinguishes between program constants 1 and static values $\top$ that are unknown at analysis time.

Iteration	1	2	3
`power(x, n − 1)`	$\bot$	1	$1 \cup \underline{X}$
`x * power(x, n − 1)`	$\bot$	$\underline{X}$	$\underline{X}$
`power(x, n) = if...`	1	$1 \cup \underline{X}$	$1 \cup \underline{X}$

The patterns showing up in the analysis can be materialized into algebraic datatypes which then form the basis for a transformation of the analyzed program to a generating extension. The algebraic datatypes come with coercion functions which are inserted at join points in the dataflow, i.e., on the branches of conditionals and at function calls.

We implemented a prototype of the framework in Haskell and manually generated a generating extension according to the results of the above analysis. It generates exactly the same programs as indicated at the end of Sec. 2. The entry point is `power_gen` and the program operates in the `Gen` monad which includes name generation and the context monad [5] for let insertion.

```
ml1 :: Monad m => (a -> m c) -> (m a -> m c)
ml2 :: Monad m => (a -> b -> m c) -> (m a -> m b -> m c)
-- datatypes
type T1 = ()                -- number one
type T2 = Ident             -- dynamic variable
data T3 = T3_1 | T3_2 Ident -- one or variable
-- coercions:
crc1_3 () = return T3_1     -- crc1_3 :: T1 -> Gen T3
crc2_3 x  = return (T3_2 x) -- crc2_3 :: T2 -> Gen T3
-- operations:             mult :: T2 -> T3 -> Gen T2
mult x T3_1     = return x
mult x (T3_2 y) = let_insert (Op2 MUL (Var x) (Var y))
-- generating extension
power_gen x n = if n == 0 then crc1_3 ()
    else ml1 crc2_3
           (ml2 mult (return x) (power_gen x (n - 1)))
```


Acknowledgments

Many thanks to Olivier Danvy, Fritz Henglein, John Hughes, Oleg Kiselyov, and Tiark Rompf for discussions and comments.

References

[1] 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*, pages 124–140, Dijon, France, 1985. Springer. LNCS 202.

[2] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In G. C. Buttazzo, editor, *EMSOFT 2004*, pages 249–258, Pisa, Italy, Sept. 2004. ACM. ISBN 1-58113-860-1.

[3] J. Launchbury and C. K. Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.

[4] T. Æ. 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 325–347, Amsterdam, 1988. North-Holland.

[5] P. Thiemann. Continuation-based partial evaluation without continuations. In R. Cousot, editor, *Proc. Intl. Static Analysis Symposium, SAS'03*, volume 2694 of *LNCS*, pages 366–382, San Diego, CA, USA, June 2003. Springer.