Partial evaluation

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

Partial evaluation basics

For program q, with **static inputs** s / **dynamic inputs** d define *partial evaluation*:

$$PE(q, s) = q_s \text{ such that } q_s(d) \equiv q(s, d)$$

Example: consider a parser with inputs g (grammar) and $\overline{c}$ (string). We want

$$PE(\text{parser}, g) = \text{parser}_g \text{ such that } \text{parser}_g(\overline{c}) \equiv \text{parser}(g, \overline{c})$$

Binding-time analysis (BTA)

Key idea: start with inputs s and d; assign each expression a *binding time*.

```
let rec pow x n =
  if n = 0 then 1
  else x * (pow x (n - 1))
```

Two key **analogies**: BTA as *type inference*; BTA as *abstract interpretation*.

Key idea: start with inputs s and d; assign each expression a *binding time*.

```
let rec pow x n =
  if n = 0 then 1
  else x * (pow x (n - 1))
```

Two key **analogies**: BTA as *type inference*; BTA as *abstract interpretation*.

Take a static environment { n $\mapsto$ 3 } and annotated program P:

```
let rec pow x n =
  if n = 0 then 1
  else x * (pow x (n - 1))
```

and β-reduce P to produce a (*more efficient*?) **specialized** program:

$$\texttt{pow}_3\ \texttt{x} = \texttt{x * (x * (x * 1))}$$

Check: $\text{pow}_3\, x \equiv \text{pow}\, x\, 3$.

Binding-time improvements

Naive specialization can produce poor results. For example, in this program

```
s + (d + 1)
```

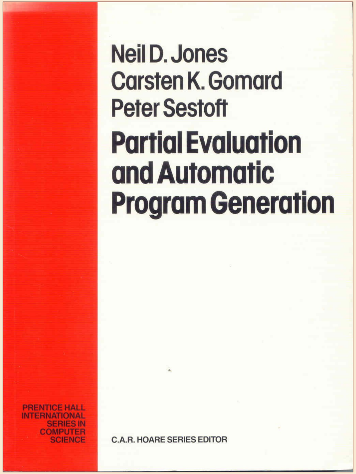
the sub-expression `d + 1` is dynamic because d is dynamic.

Binding-time improvements can bring static expressions together:

```
d + (s + 1)
```

Reading

Background reading



Partial Evaluation and Automatic Program Generation

N.D. Jones, C.K. Gomard, and P. Sestoft,
With chapters by L.O. Andersen and T. Mogensen.

Prentice Hall International
June 1993
ISBN 0-13-020249-5.

Online: `https://www.itu.dk/people/sestoft/pebook/`

Paper 1: Continuation-based partial evaluation

Partial evaluation

Reading

Continuation-Based Partial Evaluation

Julia L. Lawall
Computer Science Department
Brandeis University *
(jll@cs.brandeis.edu)

Olivier Danvy
Computer Science Department
Aarhus University **
(danvy@daimi.aau.dk)

Abstract

Binding-time improvements aim at making partial evaluation (a.k.a. program specialization) yield a better result. They have been achieved so far mostly by hand-transforming the source program. We observe that as they are better understood, these hand-transformations are progressively integrated into partial evaluators, thereby alleviating the need for source-level binding-time improvements.

Control-based binding-time improvements, for example, follow this pattern: they have evolved from ad-hoc source-level rewrites to a systematic source-level transformation into continuation-passing style (CPS). Recently, Bondorf has explicitly integrated the CPS transformation into the specializer, thus partly alleviating the need for source-level CPS transformation. This CPS integration is remarkably effective but very complex and goes beyond a simple CPS transformation. We show that it can be achieved directly by using the control operators shift and reset, which provide access to the current continuation as a composable procedure.

We automate, reproduce, and extend Bondorf's results, and describe how this approach scales up to hand-writing partial-evaluation compilers. The first author has used this method to bootstrap the new release of Consel's partial evaluator Schism. The control operators not only allow the partial evaluator to remain in direct style, but also can speed up partial evaluation significantly.

1 Introduction

Partial evaluation is a program-transformation technique for specializing programs [11, 23]. It was developed in the sixties and seventies [1, 25], drastically simplified in the eighties for purposes of self-application [24], and is now evolving both quantitatively and qualitatively. Quantitatively, partial evaluators handle more and more programming-language features — types, higher-order procedures, data structures, and so on. Qualitatively, these features need to be handled effectively. This is where *binding-time improvements* intervene [23, Chap. 12].

1.1 Binding-time improvements

The notion of binding time arises naturally in partial evaluation since source programs are evaluated in two stages: at partial-evaluation time (statically) and at run time (dynamically). The parts of the source program that can be evaluated statically are referred to as "static" and the others as "dynamic".

Obviously, the more static parts there are in a source program, the better it gets specialized. A binding-time improvement is a source-level transformation that enables more parts to be classified as static. Say that we want to partially evaluate the expression

$$x + (y - 1)$$

where we know that x is bound statically and y is bound dynamically. A naïve binding-time analysis would classify both the subtraction and the addition to be dynamic, since in each case one of the operands is dynamic. Using the associativity and commutativity laws of arithmetic, we can rewrite the expression as follows.

$$y + (x - 1)$$

The same naïve binding-time analysis would now classify the subtraction to be static (since x will be known at partial-evaluation time and 1 is an immediate constant) and the addition to be dynamic (since y will not be known until run time). By rewriting the expression, we have achieved a binding-time improvement: the same binding-time analysis classifies more expressions as static, thus enabling the same specializer to do a better job. Overall, the same partial evaluator yields a better result.

1.2 Evaluation and partial evaluation

Partial evaluation mimics evaluation — computing values of static expressions, but residualizing (*i.e.*, reconstructing) dynamic expressions, to produce the specialized program. When an expression is residualized, the continuation of the partial evaluation of its components may differ from the continuation of their evaluation. This can cause a loss of static information. Consider the Scheme expression[1]

*Waltham, Massachusetts 02254, USA. This work was initiated at the Oregon Graduate Institute of Science & Technology in summer 1993; continued at Indiana University in fall 1993; and was completed at Brandeis University. It was partially supported by NSF under grant CCR-9224375 and by ONR under grant N00014-93-1-1015.

**Ny Munkegade, 8000 Aarhus C, Denmark.

[1] The square brackets can be read as parentheses.

"Control-based binding-time improvements [...] have evolved from ad-hoc source-level rewrites to a systematic source-level transformation into continuation-passing style (CPS).

"Recently, Bondorf has explicitly integrated the CPS transformation into the specializer, thus partly alleviating the need for source-level CPS transformation. This CPS integration is remarkably effective [...] We show that it can be achieved directly by using the control operators shift and reset [...]

"The control operators not only allow the partial evaluator to remain in direct style, but also can speed up partial evaluation significantly."

Paper 2: eta expansion

Eta-Expansion Does The Trick *

Olivier Danvy Karoline Malmkjær Jens Palsberg
BRICS † MIT§
Aarhus University‡

May 1996

Abstract

Partial-evaluation folklore has it that massaging one's source programs can make them specialize better. In Jones, Gomard, and Sestoft's recent textbook, a whole chapter is dedicated to listing such "binding-time improvements": nonstandard use of continuation-passing style, eta-expansion, and a popular transformation called "The Trick". We provide a unified view of these binding-time improvements, from a typing perspective.

Just as a proper treatment of product values in partial evaluation requires partially static values, a proper treatment of disjoint sums requires moving static contexts across dynamic case expressions. This requirement precisely accounts for the nonstandard use of continuation-passing style encountered in partial evaluation. Eta-expansion thus acts as a uniform binding-time coercion between values and contexts, be they of function type, product type, or disjoint-sum type. For the latter case, it enables "The Trick".

In this article, we extend Gomard and Jones's partial evaluator for the λ-calculus, λ-Mix, with products and disjoint sums; we point out how eta-expansion for (finite) disjoint sums enables The Trick; we generalize our earlier work by identifying that eta-expansion can be obtained in the binding-time analysis simply by adding two coercion rules; and we specify and prove the correctness of our extension to λ-Mix.

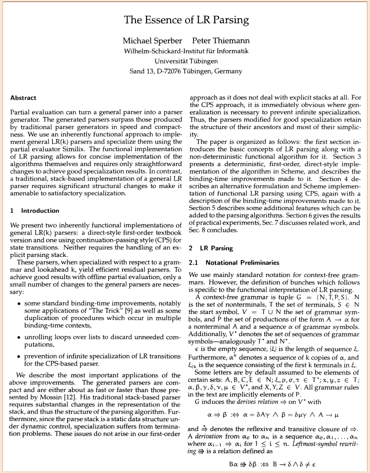
Keywords: Partial evaluation, binding-time analysis, program specialization, binding-time improvement, eta-expansion, static reduction.

"Just as a proper treatment of product values in partial evaluation requires partially static values, a proper treatment of disjoint sums requires moving static contexts across dynamic case expressions. This requirement precisely accounts for the nonstandard use of continuation-passing style encountered in partial evaluation. Eta-expansion thus acts as a uniform binding-time coercion between values and contexts, be they of function type, product type, or disjoint-sum type. For the latter case, it enables "The Trick"."

Paper 3: LR Parsing

Partial evaluation

Reading



The Essence of LR Parsing

Michael Sperber Peter Thiemann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany

Abstract

Partial evaluation can turn a general parser into a parser generator. The generated parsers surpass those produced by traditional parser generators in speed and compactness. We use an inherently functional approach to implement general LR(k) parsers and specialize them using the partial evaluator Similix. The functional implementation of LR parsing allows for concise implementation of the algorithms themselves and requires only straightforward changes to achieve good specialization results. In contrast, a traditional, stack-based implementation of a general LR parser requires significant structural changes to make it amenable to satisfactory specialization.

1 Introduction

We present two inherently functional implementations of general LR(k) parsers: a direct-style first-order textbook version and one using continuation-passing style (CPS) for state transitions. Neither requires the handling of an explicit parsing stack.

These parsers, when specialized with respect to a grammar and lookahead k, yield efficient residual parsers. To achieve good results with offline partial evaluation, only a small number of changes to the general parsers are necessary:

- some standard binding-time improvements, notably some applications of "The Trick" [9] as well as some duplication of procedures which occur in multiple binding-time contexts,
- unrolling loops over lists to discard unneeded computations,
- prevention of infinite specialization of LR transitions for the CPS-based parser.

We describe the most important applications of the above improvements. The generated parsers are compact and are either about as fast or faster than those presented by Mossin [12]. His traditional stack-based parser requires substantial changes in the representation of the stack, and thus the structure of the parsing algorithm. Furthermore, since the parse stack is a static data structure under dynamic control, specialization suffers from termination problems. These issues do not arise in our first-order approach as it does not deal with explicit stacks at all. For the CPS approach, it is immediately obvious where generalization is necessary to prevent infinite specialization. Thus, the parsers modified for good specialization retain the structure of their ancestors and most of their simplicity.

The paper is organized as follows: the first section introduces the basic concepts of LR parsing along with a non-deterministic functional algorithm for it. Section 3 presents a deterministic, first-order, direct-style implementation of the algorithm in Scheme, and describes the binding-time improvements made to it. Section 4 describes an alternative formulation and Scheme implementation of functional LR parsing using CPS, again with a description of the binding-time improvements made to it. Section 5 describes some additional features which can be added to the parsing algorithms. Section 6 gives the results of practical experiments, Sec. 7 discusses related work, and Sec. 8 concludes.

2 LR Parsing

2.1 Notational Preliminaries

We use mainly standard notation for context-free grammars. However, the definition of bunches which follows is specific to the functional interpretation of LR parsing.

A context-free grammar is tuple $G = (N, T, P, S)$. N is the set of nonterminals, T the set of terminals, $S \in N$ the start symbol, $V = T \cup N$ the set of grammar symbols, and P the set of productions of the form $A \to \alpha$ for a nonterminal A and a sequence α of grammar symbols. Additionally, V^* denotes the set of sequences of grammar symbols—analogously T^* and N^*.

ϵ is the empty sequence, $|\xi|$ is the length of sequence ξ. Furthermore, α^k denotes a sequence of k copies of α, and $\xi_{|k}$ is the sequence consisting of the first k terminals in ξ.

Some letters are by default assumed to be elements of certain sets: $A, B, C, E \in N$; $\xi, \rho, \sigma, \tau \in T^*$; $x, y, z \in T$; $\alpha, \beta, \gamma, \delta, \nu, \mu \in V^*$, and $X, Y, Z \in V$. All grammar rules in the text are implicitly elements of P.

G induces the *derives relation* $\Rightarrow$ on V^* with

$$\alpha \Rightarrow \beta \;:\Leftrightarrow\; \alpha = \delta A \gamma \wedge \beta = \delta \mu \gamma \wedge A \to \mu$$

and $\overset{*}{\Rightarrow}$ denotes the reflexive and transitive closure of $\Rightarrow$. A *derivation* from α_0 to α_n is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq n$. *Leftmost-symbol rewriting* $\Rrightarrow$ is a relation defined as

$$B\alpha \Rrightarrow \delta\beta \;:\Leftrightarrow\; B \to \delta \wedge \delta \neq \epsilon$$

"Partial evaluation can turn a general parser into a parser generator. The generated parsers surpass those produced by traditional parser generators in speed and compactness. [...]

"The functional implementation of LR parsing allows for concise implementation of the algorithms themselves and requires only straightforward changes to achieve good specialization results. In contrast, a traditional, stack-based implementation of a general LR parser requires significant structural changes to make it amenable to satisfactory specialization."

Writing suggestions

Binding-time improvements

How useful are CPS conversion and eta expansion in practice?

Are there any other generally-useful binding-time improvements

Applicability and limitations

How widely applicable is partial evaluation in practice?

What kind of performance improvements might we expect?

Compilation

Might it be practical to use partial evaluation as a compilation technique?

Demise

What happened to partial evaluation as a research field?