On Jones-Optimal Specializers: A Case Study Using Unmix

Johan Gade[1,2] and Robert Glück[2]

[1] Dept. of Mathematical Informatics, University of Tokyo
Tokyo 113-8656, Japan
jgade@acm.org
[2] DIKU, Dept. of Computer Science, University of Copenhagen
DK-2100 Copenhagen, Denmark
glueck@acm.org

Abstract. Jones optimality is a criterion for assessing the strength of a program specializer. Here, the elements required in a proof of Jones optimality are investigated and the first formal proof for a non-trivial polyvariant specializer (Unmix) is presented. A simplifying element is the use of self-application. Variations of the original criterion are discussed.

1 Introduction

When using a specializer, it is desirable to know how "strong" it is. Likewise, when engineering a new specializer, it is desirable to know when the design is "good enough". Jones proposed a criterion [6], now known as *Jones optimality*, for judging the strength of a specializer. Originally, it was presented as an aid for engineering a self-applicable partial evaluator [7], which led to some thinking this is its only value. Over the last decade, however, the notion proved useful as a standard for assessing other aspects of a specializer's strength. For example, a Jones-optimal specializer can overcome *inherited limits* [10], and it is a necessary condition for a specializer to be *translation universal* (i.e., for any compiler an interpreter exists such that the target programs produced by specializing the interpreter are as efficient as those produced by the compiler) [3].

Although the practical and theoretical implications of the criterion have become clearer, many specializers are still only *believed* to be Jones-optimal and, to our knowledge, only lambda-mix (a small partial evaluator for the lambda-calculus that does not use the usual polyvariant program-point specialization) has *formally* been shown to satisfy the criterion [15]. On the other hand, for some specializers, such as FCL-mix, it has been argued that they are *not* Jones-optimal [4]. Thus, while Jones optimality may be *plausible* in some cases, the question is usually not settled conclusively for more realistic specializers.

In this paper, the elements required in a proof of Jones optimality for non-trivial systems are investigated and Unmix, a system based on the classical offline partial evaluator Mix [8], is shown to be Jones-optimal (in a refined sense wrt a set of optimized programs, which also under suitable conditions implies Jones optimality [7]). With use of a partial evaluator that exhibits the essential

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 406–422, 2006.

features of offline partial evaluators in general, it is hoped that this study may serve as a useful guideline for investigating more realistic partial evaluators. More specifically, the contributions of this paper are to:

1. Establish that the self-applicable partial evaluator Unmix is Jones-optimal.
2. Confirm the need for variable splitting to achieve Jones optimality of specializers in languages with multiple parameters.
3. Give a framework for proving Jones optimality of more realistic specializers.
4. Explore practical issues for proving Jones optimality (e.g., textual equality vs. timed semantics and modularization of proofs by self-applicability).

The paper is organized as follows. Section 2 reviews specialization with focus on the arity of programs. Section 3 introduces Jones optimality and Sect. 4 presents the Unmix case study. Sections 5 and 6 discuss related work and conclusions. Familiarity with the basics of partial evaluation is assumed [7, Part II].

2 Program Specialization and Multi-parameter Programs

This section introduces the notation and terminology used in the paper, which should be fairly standard for readers familiar with partial evaluation. Nevertheless, to sensitize the reader to the effects of an arbitrary number of parameters, they are treated in greater detail than is usual. Understanding the subtle implications of multi-parameter programs is essential for proving Jones optimality of specializers for many untyped languages, including Unmix. A connection to the tagging problem known from specializing self-interpreters for strongly typed languages [2,9,10,16] will be made. The notation is adapted from Jones et al. [7].[1]

All programming languages in the text are assumed to be Turing-complete and untyped. In addition, every program has a fixed, but arbitrary number of parameters. Unless stated otherwise, the same language is intended as the source, residual, and implementation language of self-interpreters and specializers. We assume that the set of input values, D, includes the set of all programs, P, and all lists $(\mathtt{i}_1, \ldots, \mathtt{i}_n)$ where $\mathtt{i}_1, \ldots, \mathtt{i}_n$ are input values. The notation $\doteq$ denotes equality of partial values: Either both sides are defined and equal or both are undefined.

Definition 1 (Program evaluation, running time). *The result (if any) of* evaluating *a program* $\mathtt{p} \in P$ *with arity* $n \geq 0$ *and inputs* $\mathtt{i}_1, \ldots, \mathtt{i}_n \in D$ *is denoted by* $[\![\mathtt{p}]\!]\ \mathtt{i}_1 \cdots \mathtt{i}_n$*, and the* running time *is denoted by* $Time(\mathtt{p}, \mathtt{i}_1, \ldots, \mathtt{i}_n)$*. The partial order* $\leq_{Time}$ *for n-ary programs* p *and* q *is defined by:*

$$\mathtt{p} \leq_{Time} \mathtt{q} \ \Leftrightarrow\ \forall \mathtt{i}_1, \ldots, \mathtt{i}_n \,.\, Time(\mathtt{p}, \mathtt{i}_1, \ldots, \mathtt{i}_n) \leq Time(\mathtt{q}, \mathtt{i}_1, \ldots, \mathtt{i}_n) \qquad (1)$$

A *program specializer* is a program that given another program (the *subject* program) and some of its inputs (the *static* data), produces a *residual program* that gives the same result when evaluated with the remaining inputs (the *dynamic* data) as the subject program does when evaluated with all of its input.

[1] To make a clear distinction between evaluating programs whose input is a tuple (or more generally a list) from evaluating those with multiple inputs, the arguments are not surrounded by braces, e.g., in $[\![\mathtt{p}]\!]$ `(a, b)` there is only one input, a tuple.