Type Specialisation for the \(\lambda\)-Calculus;

or,

A New Paradigm for Partial Evaluation Based on Type Inference

John Hughes


1 Introduction

Partial evaluation is a powerful automated strategy for transforming programs, some of whose inputs are known. The classic simple example is the \textit{power} function,

\[
\text{power} \ n \ x = \text{if} \ n = 1 \ \text{then} \ x \ \text{else} \ x \times \text{power} \ (n-1) \ x
\]

which, given that \(n\) is known to be 3, can be transformed into the \textit{specialised} version

\[
\text{power}_3 \ x = x \times (x \times x)
\]

The computations on known data (static computations) are performed by the partial evaluator once and for all, and in general the resulting residual program is considerably more efficient than the original.

Over the last decade partial evaluators have developed from experimental toys into well-engineered tools. But the problem of specialising typed programs has never been satisfactorily solved. Straightforward methods produce residual programs that operate on the same types of data as the original program, but this may not be appropriate. For example, where the original program needs a sum type, the residual program may actually only use data lying in one summand. The tagging and un-tagging operations are then an unnecessary overhead; it would be better to simplify the type to the summand actually used. Such \textit{type specialisation} was identified as a 'challenging problem in partial evaluation' by Neil Jones in 1987, but there are still no really satisfactory methods for doing it.

The problem is particularly acute when the program to be specialised is an interpreter. Interpreters are universal programs which can simulate the behaviour of any other; when an interpreter is specialised to the program \(P\), the residual program is equivalent to \(P\), but is expressed in the language that the partial evaluator processes. It can be considered to be \textit{compiled code} for \(P\). But suppose the interpreter is written in a typed language: then values of every type must be represented by injecting them into one universal type, a tagged sum of all the types that can occur. When such an interpreter is specialised, the 'compiled code' produced still operates on tagged values of the universal type, and the performance benefits of compiling a typed language are lost.

Jones calls a partial evaluator \textit{optimal} if the result of specialising a self-interpreter for the language the partial evaluator processes to any program \(P\) is not only \textit{equivalent to} \(P\), it is \textit{essentially the same} as \(P\). An optimal partial evaluator can 'remove
a complete layer of interpretation'. Most partial evaluators for typed languages have not been optimal hitherto, because residual programs contain tagging and untagging operations not present in the programs being compiled.

Removing these tags carries a risk: there is a possibility that residual programs may become ill typed, in the case where a tag check would have failed. Residual programs therefore need to be type-checked. Rather than leaving this for a post-processor, we have taken it as inspiration for a new kind of partial evaluator: whereas previous ones have been, in a sense, generalised evaluators, ours is a kind of generalised type-checker. In this sense our work introduces a new paradigm for partial evaluation.

The partial evaluator we describe is the first optimal partial evaluator for the simply typed λ-calculus. It can specialise types, and can remove all unnecessary tagging and untagging operations. In particular, one universal type in a self-interpreter can be specialised to an arbitrary type in the residual program.

In the next section we informally introduce the basic ideas underlying our partial evaluator. Then we specify its behaviour formally via a set of inference rules. We go on to briefly describe the binding-time checker we use before specialisation, and a post-processor that removes trivial residual computations. Next we describe how the specialiser's inference system has been implemented, and discuss an interesting example: specialisation of an interpreter for the typed λ-calculus. Finally we discuss future improvements, describe related work, and conclude.

2 An Informal Introduction

We shall begin in this section by introducing some of the basic concepts underlying our partial evaluator, and explaining why partial evaluation by type inference is interesting.

Like many other partial evaluators, ours processes a two-level language; that is, each construct in the source program is labelled either static or dynamic, and the partial evaluator performs static computations and builds dynamic ones into the residual program. For example, the number three can appear either statically (3) or dynamically (3) — we will consistently mark dynamic constructs by underlining, as in this case. Binding times (static vs. dynamic) are reflected in the types: 3 is of type int while 3 is of type int.

Every expression gives rise to a residual expression in the specialised program. The residual expression of 3 is of course 3, while the residual expression of 3 is *, which is how we write the unique element of the one-point type. Intuitively, since 3 is known during partial evaluation we can replace it by a dummy value in the specialised program.

We shall use the notation a ~ b to mean that source expression a is specialised to residual expression b, for example 3 ~ 3 and 3 ~ *.

1 The only element of the one-point type is of course ⊥, the undefined element, but we prefer to write * to make clear that we mean the dummy value, not the bottom element of some other type. We assume a lazy semantics so that * can be freely passed as a parameter, and so on. If one prefers a strict semantics one must take * to be the defined element of a two-point type instead.