Finiteness Analysis in Polynomial Time

Chin Soon Lee*  
Datalogisk Institut, Copenhagen University, Denmark

Abstract. To achieve the termination of offline partial evaluation, it is necessary to ensure that static variables assume boundedly many values during specialization. Various works have addressed the analysis of variable boundedness, also called finiteness analysis, in the context of specializing first-order functional programs. The underlying reasoning is always: Unbounded sequences of increases in a static variable must be impossible, if they would give rise to unbounded sequences of size-decreases for some bounded-variable values.

Static analysis is used to collect a set of bipartite graphs that describe the parameter dependencies and data size changes in possible state transitions of the specializer (operating on the program). We capture the reasoning above as a condition on the graphs. This condition is decidable, but complete for PSPACE. We therefore derive a polynomial-time approximation, by considering realistic parameter size-change behaviour.

1 Introduction

1.1 Termination of Offline Partial Evaluation

Staging the computation. Partial evaluation is a general paradigm for program specialization. However, we will limit our discussion to the specialization of first-order functional programs.

Given a program and part of its input, a residual program is generated which, when given the remainder of the input, produces the same result as the original program given all of the input at once. The idea is to perform computations not depending on the missing part of the input, and thereby avoid these computations at runtime. This can be achieved by a combination of unfolding, symbolic simplification and function specialization.

Staging the partial evaluation. Offline partial evaluation implements a two-phase strategy. First, binding-time analysis (BTA) is performed to classify variables as static or dynamic, to indicate whether a variable’s values will be available during specialization. Such a classification is called a binding-time division. A division satisfies congruence if static variables are never assigned values depending on dynamic variables. Clearly, any absent input must be classified dynamic. Based on the binding-time division, each construct in the program is annotated static or dynamic, to indicate reduction or residualization.

* This research is supported by DAEDALUS, the RTD project IST-1999-20527 of the IST Programme within the European Union’s Fifth Framework Programme.
The next phase is fast, syntax-directed specialization: The binding-time of a subexpression’s evaluation is not used to decide how to reduce an expression. This leads to a lightweight second phase.

Advantages over on-line methods, which make specialization decisions based on type tags for transformation-time values, include the following.

1. **Fast specialization**: Just as calling a specialized function or program multiple times increases the savings in execution cost, specializing an annotated function or program multiple times increases the savings in transformation cost, compared to specializing unannotated code.

2. **Annotations** can be consulted to understand the specializer’s behaviour.

3. **Self application**: Specializing the specializer to an interpreter can yield an effective compiler [9]. This is an elegant approach to compiler generation, as only the interpreter has to be proven correct. Binding-time information is central to the success of self-application [5].

On the other hand, on-line partial evaluation can achieve more aggressive specialization. For example, conditional branches are not required to return values of matching binding-times, as required for syntax-directed specialization.

Two sources of non-termination. A congruent division is necessary to prevent binding-time type errors for offline partial evaluation, but it is not sufficient to guarantee a residual program.

There are two sources of non-termination for the specialization process. The first is infinite unfolding. This can be avoided by adapting methods for guaranteeing program termination. The principle: If every infinite unfolding would give rise to a sequence of static values whose sizes descend infinitely, then no infinite unfolding is possible [8,15,11]. For the rest of this paper, we will assume that finite unfolding has been ensured by marking sufficiently many function calls as dynamic. Refer to [3] for a thorough treatment of this issue.

Non-termination is also caused by accumulating parameters under dynamic control [9]. Consider specializing \( f \) below with \( d \) unknown and \( s \) equal to \([\,]\).

\[
f(d,s) = \text{if } d=s \text{ then } s \text{ else } f(d,\text{cons}(1,s))
\]

Suppose that the \( f \) call is annotated dynamic. Then the specializer would generate versions of \( f \) with \( s \) equal to \([\,], [1], [1,1], \ldots \). The standard prescription is to re-classify \( s \) as dynamic. This is known as generalization. We do not want to generalize all static computations controlled by a dynamic variable; the coding below, with \( z \) dynamic and \( xs \) static, is a useful way to coax the specializer to tabulate \( h \) with respect a finite set of values [10].

\[
g(z,xs) = \text{if } z=\text{hd}(xs) \text{ then } h(\text{hd}(xs)) \text{ else } g(z,\text{tl}(xs))
\]

These examples illustrate the concern of this article— a sufficiently precise analysis to determine that \( xs \) above is bounded during specialization, but point out that \( s \) is not bounded in the first example. Possibly unbounded variables are generalized. By repeatedly applying BTA, a safe unfolding strategy, and generalizations, the termination of specialization can be guaranteed.