Termination Analysis of Higher-Order Functional Programs

Damien Sereni¹ and Neil D. Jones²

1 Oxford University Computing Laboratory
dsereni@comlab.ox.ac.uk
2 DIKU, University of Copenhagen
neil@diku.dk

Abstract. Size-change termination (SCT) automatically identifies termination of first-order functional programs. The SCT principle: a program terminates if every infinite control flow sequence would cause an infinite descent in a well-founded data value (POPL 2001).

More recent work (RTA 2004) developed a termination analysis of the pure untyped λ-calculus using a similar approach, but an entirely different notion of size was needed to compare higher-order values. Again this is a powerful analysis, even proving termination of certain λ-expressions containing the fixpoint combinator Y. However the language analysed is tiny, not even containing constants.

These techniques are unified and extended significantly, to yield a termination analyser for higher-order, call-by-value programs as in ML’s purely functional core or similar functional languages. Our analyser has been proven correct, and implemented for a substantial subset of OCaml.

1 Introduction

Background. Termination proofs are an essential part of program verification and theorem proving. Despite its status as the canonical undecidable problem, automatic termination analysis is a useful tool.

The size-change termination principle of [10] applies to functional programs in which all datatypes are well-founded (which leaves all datatypes but integers and floats, so that natural numbers must be used in lieu of integers). It provides a fully automatic test for termination that does not rely on explicit lexicographic or other orders supplied by the user. The approach was adapted to a termination analysis of the pure untyped λ-calculus [8], using a new well-founded size ordering defined on higher-order values. Analysis of a λ-expression first produced a control flow graph together with some size-change graphs, and then applied the SCT principle.

Contributions of this paper

– A new call-by-value higher-order termination analysis of purely functional languages such as (subsets of) ML or Scheme. The first-order and λ-calculus
analyses of [10] are extended to handle higher-order values (not in [10]), and named functions, user-defined datatypes and general recursion (not in [8]).

- A depth parameter $k$ extends Shivers’ $k$-CFA [17] to trace flow of data values. This adjustable parameter steers the tradeoff between analysis precision and time complexity. The class of programs recognised as terminating strictly increases as this parameter grows.

- Depth-0 analysis encompasses the analyses of [10, 8]. Depth-1 analysis can prove termination of yet more sophisticated recursions, as well as call-by-value translations of many lazy functional programs.

- The analysis has been implemented for a subset of OCaml, and the implementation is freely available at [15].

1.1 The Core Language

Our termination analysis applies to a purely functional call-by-value language. This paper uses a very restricted core language with curried function definitions that is powerful enough to serve as an intermediate representation for most call-by-value functional languages. The abstract syntax of the core language follows in an ML-like datatype declaration:

\[
\begin{align*}
\text{Program} &= \text{FunctionDef list } \times \text{Expr} & \text{def}_1 \text{ def}_2 \cdots \text{ def}_n ; \ e \\
\text{FunctionDef} &= \text{Name } \times \text{Name list } \times \text{Expr} & \text{f} \ x_1 \cdots x_n = e \\
\text{e } \in \text{Expr} &= \text{Name} \\
& \mid \text{Name} & f \\
& \mid \text{Const of Constant} & c \\
& \mid \text{App of Expr } \times \text{Expr} & e_1 \ e_2 \\
& \mid \text{If of Expr } \times \text{Expr } \times \text{Expr} & \text{if } e \ \text{then } e_1 \ \text{else } e_2
\end{align*}
\]

A program is a list of top-level function declarations, together with an expression to evaluate in the context of these definitions. Expressions $e$ are standard and so not explained further. A core language constant may be atomic, e.g., a natural number 0 or 1; or a primitive operator, e.g., $+, −$, or, as in ML, list constructors [], :: and hd, tl, null. Numeric values are assumed well-founded so evaluation of expression $0 − 1$ will cause termination (abortion). We write $\overline{f}$ for the body $e$ of a function $f$ defined by $f \ x_1 \cdots x_n = e$.

Translation from a reasonably large OCaml subset to the core language is sketched in Section 5.1, with more details available in the companion report [16]. Thus we use OCaml syntax where convenient in examples.

The standard function \texttt{map} on lists illustrates the syntax. (The superscripts are expression labels and can be ignored for now.)

\[
\text{let rec map } f \ \text{xs } = \text{match } \text{xs } \text{with} \\
\quad [] \rightarrow [] \\
\quad x :: \text{xs } \rightarrow f \ x :: \overline{1} \text{map } f \ \text{xs}
\]

This can be translated to the core language program:

\[
\text{map } f \ \text{xs } = \text{if } \text{null } \text{xs } \text{then } [] \ \text{else } f \ (\text{hd } \text{xs}) :: \overline{1} \text{map } f \ (\text{tl } \text{xs})
\]