Dependency, Termination and Overlap Analysis of Higher-order Programs: short abstract

Neil D. Jones, DIKU, University of Copenhagen

May 7, 2003

Abstract

Some new analyses of higher-order programs are formulated and proven correct using SOS (Structural Operational Semantics). Advances over previous work: size-change analysis is extended to programs with higher-order functions; size-change graphs are computed from the program using semi-compositional SOS; correctness is easily verified because the exact and approximate semantics are closely parallel; and an SOS overlap analysis is described that identifies call sites where one function can call another (or itself) more than once with the same arguments.¹

The size-change termination analysis of [6] is based on a very simple principle: Program p terminates on every input if every infinite sequence of function calls (that follow program control flow) would cause at least one value from a well-founded domain to decrease infinitely.² Algorithmically, one constructs a data-flow graph $G_c$ for each call from program function $f$ to function $g$, with edges annotated by size-change labels ↓ or = whenever this can definitely be seen to be true (and no edge if not). The least graph set $S$ containing every $G_c$ and closed under composition is then computed. Finally, $p$ is size-change terminating in the above sense iff every idempotent graph in $S$ has an in situ decrease $x \xrightarrow{↓} x$.

Surprisingly many first-order programs are size-change terminating, even though the approach seems simple-minded at first sight because it ignores all tests appearing in $p$. The method handles Ackermann's function and needs no special treatment for general recursive programs containing mutual recursion and parameter permutation. Further, it is relatively easy to automate, its first implementation being a version programmed to aid the Agda proof assistant [4].

The computational power is known: $f$ is computable by some first-order size-change terminating program iff $f$ is multiple recursive [2]. This is a respectably large function class from a practical viewpoint. Further, many algorithms are size-change terminating, so a program manipulation system can certify as terminating programs written naturally, *e.g.*, without restriction to an annoying primitive recursive syntax.

¹Memoisation at such call sites can yield exponential efficiency increases.

²Like many good ideas, this one has been discovered more than once. Logic Programming has a similar analysis [9], and a similar construction for determining Büchi automata predated both that work and ours [10].
Overview of main results

Size-change termination is extended to higher-order programs, given in an ML- or Haskell-like “named combinator” syntax without lambdas. A further analysis is developed to to detect overlapping function calls.

1. An example program illustrates indirect recursion and “hidden” nonlinear function calls, requiring memoisation to avoid exponential running times.

2. An SOS is given for exact program execution. Sem\text{exact} represents a functional value by a “closure” \( (f \, v_1 \ldots \, v_k) \), where \( k < \text{arity}(f) \).

3. An SOS Sem\text{ex.d.flow} is given that both describes exact program execution and traces data-flow. Data-flow graphs can be extracted from any finite Sem\text{ex.d.flow} proof tree. A minor extension yields size-change graphs.

4. Next, an approximate control-flow semantics Sem\text{ap.c.flow}, similar in effect to 0-CFA \([?]\), is obtained by abstracting Sem\text{exact}. This is shown to be finitely computable and to account for all possible program control flows.

5. These two semantics are combined to yield an approximate Sem\text{ap.c.d.flow} that accounts for all possible program control flows and yields size-change graphs. The analysis is shown both correct and finitely computable.

6. An f-to-g overlap consists of two call sequences \( cs, cs' : f \rightarrow g \) that 1) are distinct, 2) yield the same argument transformations, and 3) are coupled in the sense that any computation contains \( cs \) iff it contains \( cs' \). Such nonlinear control flow will definitely cause computational redundancy if encountered, and is particularly expensive if \( g \) calls \( f \) again.

7. The problem of detecting exact overlap is seen to be undecidable. A final approximate program analysis Sem\text{ap.overlap} is given that will detect systematic overlap if it is present.

8. It is shown that any HOPR (higher-order primitive recursive) program will be certified as terminating by the new method. Consequence: Function \( f \) is computable by some higher-order size-change terminating program iff it is definable in Gödel’s System T, i.e., iff it is \( \epsilon_0 \)-recursive.

An example

Consider the following second-order HOPR program with call sites labeled 1–6:

Types:

- \( \text{F} : * \rightarrow *, \ 1 \) : \( * \rightarrow * \), \( \text{S} : * \rightarrow * \rightarrow * \), \( \text{N} : (\text{\texttt{\textbullet}} \rightarrow \text{\textbullet}) \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \)

Definitions:

- \( \text{F x} = 1 : \text{S x x} \)
- \( \text{S t} = \text{if } t \rightarrow \text{ then } 2: \text{Id else } 3: \text{N (4: S(t-1))} \)
- \( \text{N r i} = \text{S: r(i) + 6: r(i+1)} \)
- \( \text{Id z} = z \)
Analyses:

The full paper contains details of the SOS rule systems, their usage to perform the following program analyses, and some soundness proofs.

1. Control-flow analysis, net effects of the defined functions:
   \( F * \) \( \rightarrow * \) and \( \langle \text{Id} * \rangle \) \( \rightarrow * \), each maps a base value into a base value.
   \( \langle S * \rangle \) \( \rightarrow * \), function \( S \) maps two base values into a base value.
   \( \langle N \langle S * \rangle \rangle \) \( \rightarrow * \), \( N \) maps a function and a base value into a base value. The function can only be the result of applying \( S \) to a single base value.

2. Control flow analysis, effects of the function calls:

Call site 4 is not “really” recursive since the \( S \) argument list is incomplete. On the other hand, call sites 5 and 6 both complete the \( S \) argument list, so 3 : \( S \rightarrow N \), 5 : \( N \rightarrow S \) and 3 : \( S \rightarrow N \), 6 : \( N \rightarrow S \) complete recursive loops.

3. Size-change graphs for call sites in the two loops:

4. Call-duplication analysis:

Consider call sequences 3536 and 3635, both taking \( S \rightarrow N \rightarrow S \rightarrow N \rightarrow S \). These are coupled: if either call sequence is performed in a computation, then so must the other. Further, they have the same effect: that \( \langle S t \rangle \) calls \( \langle S t \rangle (i+2) \). Thus the program has a hidden nonlinearity, caused by the interaction of recursion and double usage of parameter \( x \).

The net effect is call duplication, causing the program to run for time of order \( \Omega(2^n) \) on input \( x \) if, say, call-by-value is used. This can be reduced to a small polynomial by memoisation.

\[ \text{Remark: By 1, the * parameter \( x \) must have the form of a closure \( \langle s t \rangle \). In graphs \( G_5, G_7, G_8 \) this is written vertically, using \( - \) and \( \langle \) and \( \rangle \), respectively.} \]
References


