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 $\downarrow$ or $=$ whenever this can definitely be seen to be true (and no edge if not). The least graph set $\mathcal{S}$ 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 $\mathcal{S}$ has an *in situ* decrease $\mathtt{x} \stackrel{\downarrow}{\rightarrow} \mathtt{x}$.

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 implemention 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.

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

[2] Like many good ideas, this one has been discovered more than once. Logic Programming has a similar analysis [9], and a similar construction for determinising Büchi automata predates 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_{exact} represents a functional value by a "closure" $\langle \mathtt{f}\ v_1 \ldots v_k \rangle$, where $k < \mathit{arity}(\mathtt{f})$.
3. An SOS $\mathrm{Sem}_{ex\text{-}d\text{-}flow}$ is given that both describes exact program execution and traces data-flow. Data-flow graphs can be extracted from any finite $\mathrm{Sem}_{ex\text{-}d\text{-}flow}$ proof tree. A minor extension yields size-change graphs.
4. Next, an approximate *control-flow semantics* $\mathrm{Sem}_{ap\text{-}c\text{-}flow}$, similar in effect to 0-CFA [?], is obtained by abstracting Sem_{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 $\mathrm{Sem}_{ap\text{-}c\text{-}d\text{-}flow}$ 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 $cs, cs' : \mathtt{f} \to \mathtt{g}$ 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 $\mathrm{Sem}_{ap\text{-}overlap}$ 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 ϵ_0-recursive.

An example

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

```
Types:
       F : • → •, Id : • → •,
       S : • → • → •,
       N : (• → •) → • → •
Definitions:
       F x    =  1: S x x
       S t    =  if t=0 then 2: Id else 3: N (4: S(t-1))
       N r i  =  5: r(i) + 6: r(i+1)
       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.

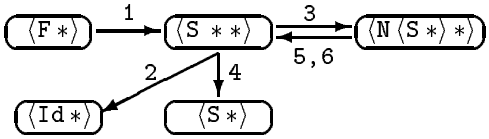
1. **Control-flow analysis, net effects of the defined functions:**

   $\langle \mathtt{F}\, * \rangle \mapsto *$ and $\langle \mathtt{Id}\, * \rangle \mapsto *$, each maps a base value into a base value.

   $\langle \mathtt{S}\ *\, * \rangle \mapsto *$, function S maps two base values into a base value.

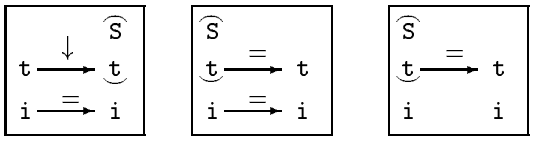
   $\langle \mathtt{N}\, \langle \mathtt{S}\, * \rangle\, * \rangle \mapsto *$, N maps a function and a base value into a base value. The function can only be the result of applying S to a single base value.

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 $3 : \mathtt{S} \to \mathtt{N}, 5 : \mathtt{N} \to \mathtt{S}$ and $3 : \mathtt{S} \to \mathtt{N}, 6 : \mathtt{N} \to \mathtt{S}$ complete recursive loops.

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



   $G_3 : \mathtt{S} \to \mathtt{N}$ $G_5 : \mathtt{N} \to \mathtt{S}$ $G_6 : \mathtt{N} \to \mathtt{S}$

4. **Call-duplication analysis:**

   Consider call sequences 3536 and 3635, both taking $\mathtt{S} \to \mathtt{N} \to \mathtt{S} \to \mathtt{N} \to \mathtt{S}$. These are *coupled*: if either call sequence is performed in a computation, then so must the other. Further, they have the same effect: that $\langle \mathtt{S}\, t\, i \rangle$ calls $\langle \mathtt{S}\, (t-2)\, (i+1) \rangle$. Thus the program has a hidden nonlinearity, caused by the interaction of recursion and double usage of parameter r.

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

[3] **Remark:** By 1, the N parameter r must have the form of a closure $\langle \mathtt{S}\, \mathtt{t} \rangle$. In graphs G_3, G_5, G_6 this is written vertically, using $\frown$ and $\smile$ for $\langle$ and $\rangle$, respectively.

References

[1] Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97, Lecture Notes in Computer Science* vol. 1232, pp. 157–171. Springer, 1997.

[2] Amir M. Ben-Amram. General Size-Change Termination and Lexicographic Descent. In *The Essence of Computation: Complexity, Analysis, Transformation.*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2002.

[3] Wei-Ngan Chin, Siau-Cheng Khoo, and Tat-Wee Lee. Synchronisation analysis to stop tupling. In *Programming Languages and Systems (ESOP'98)*, pages 75–89, Lisbon, 1998. Springer LNCS 1381.

[4] Catarina Coquand. The interactive theorem prover Agda. http://www.cs.chalmers.se/~catarina/agda/, 2001.

[5] Olin Shivers. Control-flow analysis in Scheme. Proceedings of PLDI, the SIGPLAN '88 Conference on *Programming Language Design and Implementation*, June 1988.

[6] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM Press, January 2001.

[7] Neil D. Jones and Arne J. Glenstrup. Program Generation, Termination, and Binding-time Analysis. In *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering GPCE'02*, volume 2487 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2002.

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall. Download accessible from `www.diku.dk/users/neil`, 1993.

[9] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Termilog: A system for checking termination of queries to logic programs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, Jun 22–25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 444–447. Springer, 1997.

[10] Aravinda Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[11] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th Int. Symposium, SAS '97, Paris, France, Sep 8–19, 1997, Lecture Notes in Computer Science*, vol. 1302, pp. 160–171. Springer, 1997.