Analysis of Typed Intermediate Representations

(Really Short Abstract)[*]

Nevin Heintze[†]

May 30, 1997

I will give a preliminary report on an implementation of linear-time control-flow analysis (CFA) for the current LAMBDA language of the SML/NJ compiler (see Zhong Shao's paper in this workshop [3]). The motivation of this work is four-fold. First, to provide a testbed for the new control-flow analysis algorithm. Second, to understand how the presence of types in an intermediate representation interacts with analysis — Can the types be exploited by the analysis? Are there new applications of the analysis e.g. what if I know something about the possible bindings of type variables? Third, to support applications of analysis (e.g. array-bounds optimizations) in the current version of SML/NJ. Forth, to provide a (somewhat) generic implementation of SBA-like analysis for SML. Most SML compilers implement some form of CFA — usually this is just a simple known/escaping functions analysis, because standard cubic-time CFA is considered too expensive. Early results from the new algorithm indicate that it is as fast as these simpler algorithms. It therefore seems possible to build an off-the-shelf analysis component for SML compiler implementors that costs the same as the usual known/escaping functions analysis, but provides much more information.

The Core Algorithm

The core algorithm of the implementation is the linear-time CFA algorithm recently developed by myself and David McAllester [1]. Specifically, given a bounded-type program, this algorithm builds a directed graph whose transitive closure gives exactly the results of the standard (cubic-time) control-flow analysis algorithm. This algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. However, compiler applications often do not use this "all-calls for all call-sites information". The following CFA applications are more typical:

- k-limited CFA[1]: for each call-site, list the functions called if there are less than k of them, and otherwise output "many".

[*]There is no full paper yet – this is just a brief summary of what I plan to talk about at the workshop.

[†]Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, nch@research.bell-labs.com.

[1]For some fixed k.

- called-once analysis: identify all functions called from only one call-site.
- effects analysis: use the CFA information to determine the side-effecting expressions in a program.

All of these applications run in linear-time using the new algorithm, while returning the same results as those obtained using the cubic-time algorithm.

Extensions: Set-Based Analysis

The basic CFA algorithm deals only with a simple version of the lambda calculus (with just abstraction and application). This must be extended to deal with recursion, records and (recursive) datatypes, assignment, exceptions, etc. [1] describes such extensions of the basic algorithm in a manner very closely related to set-based analysis (SBA) [2]. One difference is that the treatment of recursive datatypes in this extension of the linear-time algorithm is technically less accurate than in set-based analysis. An important question is how significant the difference is in practice.

Open Questions

- Are there new ways of using an SBA-like analysis in the context of a typed intermediate language? For example, is it useful to know about the bindings of type variables?
- How do we support analysis in the context of separate compilation? The graph-based approach has advantages here — it is relatively simple to project the graph wrt a set of "relevant" nodes. The typed intermediate representation also has advantages — we can use the types to bound the behavior of unknown values.
- Can we build a generic implementation of the linear-time algorithm that is easy to port to a variety of different typed intermediate representations? Are there differences between these representations that are significant from an analysis perspective?

References

[1] N. Heintze, D. McAllester, "Linear-time Subtransitive Control Flow Analysis", *1997 ACM Conference on Programming Language Design and Implementation*, to appear; http://www.cs.cmu.edu/afs/cs/user/nch/ftp/linear-cfa.ps.

[2] N. Heintze, "Set-Based Analysis of ML Programs", *ACM Conference on Lisp and Functional Programming*, pp 306–317, 1994.

[3] Z. Shao, "An Overview of the FLINT/ML Compiler" this proceedings; http://www.cs.bc.edu/ muller/TIC97/Shao.ps.