Compiling Functional Languages with Flow Analysis

Suresh Jagannathan and Andrew Wright
NEC Research Institute, Princeton, NJ 08540
{suresh,wright}@research.nj.nec.com

1 Goals

We argue for the use of aggressive interprocedural flow analysis to guide optimizations for higher-order languages like Scheme [2] and ML [6]. Functional languages provide abstraction through first-class procedures and abstract datatypes. Without sophisticated interprocedural analysis, compilers must make overly conservative assumptions about how abstractions are used in the program. These assumptions often lead to poor performance. We present a strategy for building high-performance implementations that relaxes these assumptions and provides a framework upon which useful optimizations can be built. We describe two experiments which provide some evidence that this approach is viable.

2 Motivation

Our strategy for building high performance implementations of higher-order languages is to use aggressive interprocedural flow analysis to drive a variety of global program optimizations, and to use more traditional dataflow analyses to enable local optimizations. In our view, these two classes of optimizations are not completely orthogonal. Useful global optimizations are likely to make local optimizations more effective by exposing optimization opportunities that would otherwise be missed. They are also likely to facilitate program transformations whose results are more amenable for optimization by local analyses.

Among the global optimizations we have considered include:

1. *Specialized calling protocols.* Well-written functional programs consist of many small procedures. If all call-sites of a procedure P are known, then a specialized calling protocol can be used for P. The calling protocol may allocate P's closure on the stack or in registers. If P is suitably small, its body may be inlined at the call sites, eliminating the call altogether.

2. *Specialized data representations.* Most functional language implementations assume all values are uniformly represented in a single machine

word. Objects that do not fit in a single word (e.g., floating-point numbers, closures, records, lists) or objects whose size cannot be determined by the compiler are typically *boxed*, i.e., allocated on the heap and referenced via a pointer. If all references to a boxed object are known, the object may be unboxed, and references to the object replaced with the object itself.

3. *Specialized primitive operations.* Functional languages are type-safe. Type-safety ensures that the behavior of both correct and incorrect programs is fully explained by the language semantics. In other words, programs cannot "dump core." Type safety requires that many primitive operations check the validity of their arguments at run-time. Even statically typed languages require run-time checks for projections from sum types. If all arguments to which a primitive is applied can be determined to be a member of the type expected by the primitive, the primitive need not include a run-time check.

Interprocedural analysis is essential to build optimizations of the kind listed above because these optimizations depend on global properties of the program. Once procedures have been inlined, specialized closure representations selected, values unboxed, and run-time checks removed, conventional intraprocedural optimizations that use cheaper analyses are likely to have more effect.

We believe that our compilation strategy is viable because recent developments in flow analysis research are yielding practical analyses. These new analyses can handle large programs with reasonable time and space, and the information they produce is sufficiently precise to drive useful optimizations [3, 4, 5, 7]. We expect our approach to work best for mostly functional programs that make extensive use of higher-order procedures, polymorphism, and data abstraction.

3 Flow analysis

In its simplest form, a flow analysis for a higher-order language determines the sets of values that variables may be bound to and the sets of values that expressions may yield [8]. Some implementations of higher-order languages incorporate this kind of simple analysis, but optimizations based on it have had only limited success. Because these *monovariant* analyses fail to distinguish the behavior of a procedure at one call-site from its behavior at another, the information they compute may be too imprecise to support effective optimizations.

Polyvariant [1] analyses distinguish different uses of a procedure based on some notion of abstract context. An abstract context identifies a set of program states. Analyses whose abstract contexts describe smaller sets of program states tend to yield more precise information than analyses whose abstract contexts describe larger sets. At one extreme, a monovariant analysis can be regarded as using a single abstract context that represents all possible program states. At the other extreme, an (uncomputable) analysis that represented each program state with a unique abstract context would yield exact information.

The tradeoff between precision and cost in polyvariant flow analysis is unclear. There is no direct correlation between the granularity of abstract contexts and the precision of the analysis. Too many abstract contexts can lead to redundant recomputation that increases the cost of the analysis without increasing its precision. But, too few abstract contexts can lead to overly conservative estimates of the set of procedures called at each call-site, which in turn means the analysis must investigate more abstract procedure calls. Finding a polyvariant analysis that provides precise information at reasonable cost and is robust over different programming styles and paradigms remains a challenging research problem.

4 Optimizations

To illustrate the effectiveness of flow analysis as a compilation tool, we summarize two experiments drawn from our own work.

Our first experiment investigated eliminating run-time type checks from Scheme programs using information computed by a polyvariant flow analysis [4]. Consider a particular application of a primitive p to arguments v_1, v_2, ..., v_n. In general, p requires that each of its arguments belong to a certain type, and in the absence of any optimizations, the compiled code for applications of p will usually include run-time checks to ensure type-safety. A flow analysis of the program containing this application computes approximate sets of values for each of the v_i. If the set of values the analysis computes for v_i is a subset of the type p requires for its i^{th} argument, then the run-time check on the i^{th} argument can safely be omitted from this application. We have found that run-time check elimination based on a polyvariant flow analysis can eliminate the majority of run-time checks in realistic programs.

Our second experiment investigated inlining in Scheme programs [5]. Inlining is an especially important optimization in functional languages because procedures are the primary means by which data and control abstractions are built. By eliminating procedure call overhead, other optimizations and simplifications may become apparent.

Conventional inlining optimizations are typically based on syntactic criteria that use *ad hoc* heuristics to determine the procedures that are candidates for inlining, and the call sites where inlining can be performed. Such heuristics provide only a weak capability to identify inlining sites. Procedures used in higher-order ways are rarely inlined because the heuristics are too weak to identify their call sites. In many systems, a given procedure is inlined at either all of its call sites or none of them, because the heuristics are unable to distinguish the profitability of inlining at one call-site from another.

By using flow analysis as the basis for an inlining optimization, however, we derive a very different methodology. The output of a polyvariant flow analysis can be used to disambiguate different uses of a procedure at different call sites. Thus, inlining can be highly selective. Flow analysis is also well-suited to tracking the movement of higher-order procedures. An inlining optimization can take advantage of this property by inlining procedures that were passed in a higher-order manner. Flow analysis can also yield a more precise measure of a procedure's cost. The cost defined for inlining a procedure P at some call site C should be some approximation of P's size when specialized at C. The specialized copy may be smaller if conditional branches can be computed statically based on the abstract context in which C is evaluated. Here also, we have found that interprocedural analysis exposes many opportunities for inlining that are unlikely to have been discovered otherwise.

5 Conclusion

We propose that compilers for higher-order languages use flow analysis to drive global optimizations. Flow analysis is becoming a mature technology. A compilation framework that combines global analyses and a suite of interprocedural optimizations, along with more traditional dataflow analyses and local optimizations, offers a sound basis upon which high-performance implementations for higher-order languages can be built.

References

[1] M.A. Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

[2] William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[3] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 306–317, 1994.

[4] Suresh Jagannathan and Andrew Wright. Effective Flow-Analysis for Avoiding Runtime Checks. In *Second Interntational Symposium on Static Analysis*, pages 207–225, 1995. Springer-Verlag LNCS 983.

[5] Suresh Jagannathan and Andrew Wright. Flow-directed Inlining. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1996.

[6] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[7] Manuel Serrano and Marc Feeley. Storage Use Analysis and its Applications. In *Proceedings of the ACM International Conference on Functional Programming*, May 1996.

[8] Olin Shivers. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, 1991.