Abstract. Flow analysis is a potentially very useful analysis for higher order functional languages, but its practical application has been slow in coming, partially hindered by shortcomings of the current analysis techniques. Among these are the limited precision, long analysis times, incompatibility with separate compilation, inapplicability to untyped languages and sensitivity to program structure associated with various earlier formulations.

We address these shortcomings with an approach based on a novel type system integrating recursive types, subtyping with liberal union types, and system F style polymorphism, in a constraint based formulation. The type system is “soft” in the sense that it does not reject programs. It is also not decidable, but there is a broad spectrum of sound and terminating, though incomplete, algorithms whose precision can be characterized by restricting the substitutions used for instantiation of type schemes. We give an inference algorithm which simplifies constraints incrementally, rather than postponing all constraint solving until all of the constraints are generated. In this way we hope to avoid in practice the exponential growth in the size of constraint sets caused by repeated instantiation of unsimplified type schemes. Type schemes are also a suitable representation for communication the analysis result over module boundaries, making separate compilation feasible.

Keywords: Flow analysis, type systems, constraints, polymorphism, polyvariance, inference algorithms, functional languages.

1 Introduction

Flow analysis\(^1\) of higher order functional languages determines, for each program point (or PP) which values this PP might take on during the execution of the program (in a functional language, the subexpressions of the program are typically considered PPs). The information gained is very useful for a number of important optimizations. For instance, the present author has achieved speed-ups of over a factor of three when compiling a lazy functional language [2].

Wide spread adoption of flow analysis has been slow in coming due to limitations in the current techniques. The problems include:

\(^1\) What we call flow analysis, and very similar analyses, have been called different things by different authors. Examples include set based analysis [3], closure analysis [8, 9], or control flow analysis [10, 11].
Precision. Much of the early work on flow analysis of higher order languages is monovariant, i.e. based on finding a single approximation to the possible execution-time values of each PP (such approximations are often called abstract values since most of the work in flow analysis has been more or less based on abstract interpretation) such that certain consistency conditions are satisfied.

This means that, for each $\lambda x.e$ in a program, the abstract value of $x$ approximates all arguments that $\lambda x.e$ may be applied to during execution, and the abstract value of every application where the function part may evaluate to $\lambda x.e$ includes the abstract value of $e$. This may lead to a severe loss of precision, especially in large programs where the same functions are applied in many places.

Performance. Increasing the precision of these early analyses by using more than one abstract value for each PP (generally called polyvariance as opposed to monovariance) has proved rather difficult because of combinatorial explosion. One popular approach is based on bounded length call strings which essentially capture the $k$ topmost return addresses in the stack. The number of different call strings, and thus of different abstract values to keep track of for each PP, grows exponentially with the length $k$ of the call strings. Also, call strings are of little help for recursive functions since they typically have statically unbounded depths in their recursion.

An alternative method is to clone (make several copies of) the functions in the program and run a monovariant analysis on the resulting program (see [3]). This approach has the same limitations with respect to combinatorial explosion as the previous one, but can be used to sidestep the problems with recursion (since clones are only made for nonrecursive calls).

Program structure. Both call string based and cloning based analyses are very sensitive to program structure. Adding just another layer of function calls can lead to a drastic reduction in precision.

Separate compilation. Most current techniques depend on the entire program being available for analysis. The root of this problem is related to the bad precision — since different calls to the same functions may share the same abstract value, the calls have to be analyzed simultaneously. Also, there is often no natural way to represent the result of analyzing part of a program.

Incompatibility. Type based approaches, which suffer less from the above problems, are restricted to typable programs, and thus to typed languages, and, since that is what one really wants to analyze, to typed intermediate languages. For this reason, type based analyses have not been immediately applicable to untyped languages such as Scheme and Erlang.

This paper reports on work in progress addressing these problems using a novel type-based analysis with the following features:

Subtypes. Subtype constraints are capable of capturing the direction of the flow of data in the program.

Soft typing. All programs are typable in our system. This means that well-typed programs (i.e. programs that can be successfully analyzed), can still