Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis

Tobias Gedell[1], Jörgen Gustavsson[2], and Josef Svenningsson[1]

[1] Department of Computing Science,
Chalmers University of Technology and Göteborg University
{gedell, josefs}@cs.chalmers.se
[2] Spotfire, Inc.

Abstract. There are a number of choices to be made in the design of a type based usage analysis. Some of these are: Should the analysis be monomorphic or have some degree of polymorphism? What about subtyping? How should the analysis deal with user defined algebraic data types? Should it be a whole program analysis?

Several researchers have speculated that these features are important but there has been a lack of empirical evidence. In this paper we present a systematic evaluation of each of these features in the context of a full scale implementation of a usage analysis for Haskell.

Our measurements show that all features increase the precision. It is, however, not necessary to have them all to obtain an acceptable precision.

1 Introduction

In this article we study the impact of polymorphism, subtyping, whole program analysis and accurate data types on type based *usage analysis*. Usage analysis is an analysis for lazy functional languages that aims to predict whether an argument of a function is used at most once. The information can be used to reduce some of the costly overhead associated with call-by-need and perform various optimizing program transformations. The focus of this paper is however solely on improving the precision of usage analysis, not on its uses.

Polymorphism. Polymorphism is the primary mechanism for increasing the precision of a type based analysis and achieving a degree of context sensitivity.

Previous work by Peyton Jones and Wansbrough has indicated that polymorphism is important for usage analyses. Convinced that polymorphism could be dispensed with they made a full scale implementation of a completely monomorphic usage analysis. However, it turned out that it was "almost useless in practice" [WPJ99]. They drew the conclusion that the reason was the lack of polymorphism. In the end they implemented an improved analysis with a simple form of polymorphism that also incorporated other improvements [Wan02]. The resulting analysis gave a reasonable precision but there is no evidence that polymorphism was the crucial feature.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 200–216, 2006.

Studies of other program analyses have come to a different conclusion about polymorphism. On example is points-to analysis for C for which several studies have shown that monomorphic analyses [FFA00, HT01, FRD00, Das00, DLFR01] give adequate precision for the purpose of an optimizing compiler [DLFR01]. Moreover, extending these analyses with polymorphism seem to have only a moderate effect [FFA00, DLFR01].

Point-to analysis may not be directly relevant for usage analysis but it still begs the question of how much polymorphism really can contribute to the precision of an analysis. One of the goals of this paper has been to shed some light on this question.

Subtyping. Another important feature in type based analysis is subtyping. It provides a mechanism for approximating a type by a less informative super type. This gives a form of context sensitivity since a type may have different super types at different call sites. It also provides a mechanism for combining two types, such as the types of the branches of an if expression, by a common super type. Thus, the effects of subtyping and polymorphism overlap.

This raises a number of questions. Does it suffice with only polymorphism or only subtyping? How much is gained by having the combination?

Whole program analysis. Another issue that also concerns context sensitivity is whole program analysis versus modular program analysis. A modular analysis which considers each module in isolation must make a worst case assumption about the context in which it appears.

This will clearly degrade the precision of the analysis. But how much? Is whole program analysis a crucial feature? And how does it interact with the choice of monomorphism versus polymorphism?

Data types. Another important design choice in a type based analysis is how to deal with user defined data types. The intuitive and accurate approach may require that the number of annotations on a type is exponential in the size of the type definitions of the analyzed program. The common solution to the problem is to limit the number of annotations on a type in some way, which can lead to loss of precision. The question is how big the loss is in practice.

Contributions. In order to evaluate the above features, we have implemented a range of usage analyses:

- With different degrees of polymorphism (Section 3)
- With and without subtyping (Section 4)
- Using different treatments of data types (Section 5)
- As whole program analyses and as modular analyses (Section 6)

All analyses have been implemented in the GHC compiler and have been measured with GHC's optimizing program transformations both enabled and disabled. We present figures summarizing (the arithmetic mean of) the effectiveness of each of the different features. More detailed figures for each of the programs we've analyzed can be found in the first authors licentiate thesis [Ged06].