Abstract

This thesis is concerned with the resource consumption of lazy functional languages. It touches upon two aspects: how to reason about the space-safety of program transformations, and how to apply usage analysis for compiler optimisation. The thesis is a collection of articles.

In the first paper we study the notion of *space improvement*. We say that that a program fragment is *space improved* by another if and only if when we replace the former by the latter in any whole program the space behaviour is improved. We will refer to the induced equivalence as *space equivalence*.

We show that many of the extensional equivalences that lazy functional languages enjoy carry over as space equivalences, and we demonstrate that the space improvement theory can be used to show space properties of some interesting small programs. We also show that many extensionally equivalent program fragments are (sometimes surprisingly) not space equivalent by giving examples of whole programs where the asymptotic space behaviour changes if one replaces a program fragment by the another extensionally equivalent one.

An example of a transformation that is not a space equivalence in general is the *inlining* of function calls, i.e., replacing a function call with a copy of the body of the function with the arguments substituted for the formal parameters. In the second paper of thesis we study a class of automatic methods called *usage analyses* which can infer that an argument to a function is used at most once, and show that usage analyses can be used to guarantee the work and space safety of inlining.

Another application of usage analysis is compiler optimisation. In particular usage analysis can be used to avoid unnecessary closure updates. In the third paper of the thesis we present a usage analysis for this purpose which also provides additional information which can be used to optimise the bookkeeping of updates by avoiding unnecessary *update marker checks*.

In the fourth paper of the thesis we present a context sensitive usage analysis based on bounded usage polymorphic types. To implement the analysis efficiently we introduce a new form of constraint and in the fifth paper we show how the new form of constraints can be solved. The techniques can be applied not only to usage analysis but also to similar analyses. As an example of such, we present a *flow analysis* with flow subtyping, flow polymorphism and flow-polymorphic recursion, and show how it can be implemented in $O(n^3)$ time where n is the size of the explicitly typed program.

Keywords: lazy functional languages, equational theory, improvement theory, garbage collection, space use, space-equivalence, space-safety, work-safety, inlining, program analysis, usage analysis, sharing analysis, context sensitive, constraint solving.