Abstract

In this PhD Dissertation we develop methods for proving contextual equivalence and termination.

Contextual equivalence. It is common to define two programs to be contextually equivalent if they have the same termination behavior in all closing contexts. We analyse equivalence in a functional language with recursive and polymorphic types and extended with constructs to dynamically allocate and update higher-order store. The aim is to give a sound characterization that eases proofs of contextual equivalence in many cases.

In the thesis we present three papers on contextual equivalence. All three share a common setup: They are based on an FM-denotational model with an admissible Kripke-style logical relation on top of a universal recursive domain. The methods give ways to express why two programs are expected to be equivalent via definitions of local parameters. Such definitions of local parameters express the intuition for why two programs are equivalent and are essentially the only non-trivial parts in a proof of equivalence. The combination of expressible parameters and a recursive domain makes it, however, non-trivial to establish the existence of the relations in the first place. This has required some new ideas. Our work is inspired from the work of Benton and Leperchey and extends earlier research on equivalence proofs in several ways. We extend the language to encompass recursive and polymorphic types and dynamic allocation of higher order store. Further we refine the definition of parameters and also taylor the definition to the extended language. The additional features in our language add some extra significant complications to the understanding of equivalence as well as to the denotational interpretation. Our method is the first proof method for contextual equivalence based on a logical relation over a denotational semantics for a language with recursive types and dynamic allocation of references of any type.

The first paper gives the general setup. The second paper gives a relationally parametric interpretation of polymorphic types. This is not quite as general as we would like; we restrict the type of references so that the type must be closed. It is to our knowledge the first relationally parametric model for higher-order store and polymorphic and recursive types. The third paper is primarily concerned with refining the definition of parameters.

Termination analysis. The last paper develops a sound and fully-automated algorithm to show that evaluation of a given untyped λ-expression will terminate under call-by-value. The "size-change principle" from first-order programs is extended to arbitrary untyped λ-expressions in two steps. The first step suffices to show call-by-value termination of a single, stand-alone λ-expression. The second suffices to show termination of any member of a regular set of λ-expressions, defined by a tree grammar.