Reasoning about Contextual Equivalence: From Untyped to Polymorphically Typed Calculi

David Sabel, Manfred Schmidt-Schauß, and Frederik Harwath

Goethe-University Frankfurt am Main
{sabel,schauss,harwath}@informatik.uni-frankfurt.de

This paper describes a syntactical method for contextual equivalence in polymorphically typed extended lambda-calculi based on techniques developed earlier for an untyped setting. Our specific calculus closely corresponds to an intermediate language for a Haskell compiler. It has letrec, data constructors, case-expressions, seq, and recursive types. There are an untyped and a typed variant, where the typed language is a subset of the untyped language. Evaluation of typed expressions is performed by evaluating their type erasure.

The previously investigated untyped setting comprises the following: a small-step operational semantics as a normal-order reduction to weak head normal forms (WHNF), the definition of contextual equivalence as $s \sim t$ if for all contexts C: $C[s]$ reduces to a WHNF iff $C[t]$ does, a context lemma restricting the contexts to reduction contexts and a diagrammatic method to show correctness of program transformations. These techniques and correctness results are transferred to the typed setting in this paper. Since program transformations in the typed setting may depend on the types of the subexpression which is modified, we use type-labels for all subexpressions of the typed expressions to fix the type. Polymorphic types are only visible at bindings in the letrec, all other types are monomorphic. The types of variable-occurrences may be instances of their original type, provided it is polymorphic. Type labels within an expression must be consistent, which is enforced by type constraints and a well-scopedness condition. Since the typed language can be seen as a subset of the untyped one – after erasing the type labels –, which applies to expressions and contexts, it is easy to lift untyped equivalences into the typed language. This already shows a considerable subset of transformations to be correct.

However, since there are more correct and quite interesting program transformations, we have to lift the proof methods into the typed setting. The key technique to bridge this gap is to introduce a typed normal order reduction which operates on type labeled expressions and requires to define the type-inheritance after a reduction step, where a technical innovation is the introduction of the quantifier-distribution after the (llet)-rule. Typed normal order reduction induces the same notion of convergence as normal order reduction on type-erased expressions, i.e. both reduction relations are interchangeable for reasoning about contextual equivalence. This allows us to prove a typed context lemma, and to apply the diagrammatic method in the typed calculus. We show the correctness of the transformation (IdL) that replaces $\texttt{case}_{\texttt{List}}\ s\ \texttt{of}\ (\texttt{Nil} \texttt{->} \texttt{Nil})\ ((\texttt{Cons}\ x\ xs) \texttt{->} (\texttt{Cons}\ x\ xs))$ by $s :: [T]$, which is not correct in the untyped setting.