Relational Reasoning for Recursive Types and References

Nina Bohr and Lars Birkedal

IT University of Copenhagen (ITU)
{ninab, birkedal}@itu.dk

Abstract. We present a local relational reasoning method for reasoning about contextual equivalence of expressions in a λ-calculus with recursive types and general references. Our development builds on the work of Benton and Leperchey, who devised a nominal semantics and a local relational reasoning method for a language with simple types and simple references. Their method uses a parameterized logical relation. Here we extend their approach to recursive types and general references. For the extension, we build upon Pitts' and Shinwell's work on relational reasoning about recursive types (but no references) in nominal semantics. The extension is non-trivial because of general references (higher-order store) and makes use of some new ideas for proving the existence of the parameterized logical relation and for the choice of parameters.

1 Introduction

Proving equivalence of programs is important for verifying the correctness of compiler optimizations and other program transformations. Program equivalence is typically defined in terms of *contextual equivalence*, which expresses that two program expressions are equivalent if they have the same observable behaviour when placed in any program context C. It is generally quite hard to show directly that two program expressions are contextually equivalent because of the universal quantification over all contexts. Thus there has been an extensive research effort to find reasoning methods that are easier to use for establishing contextual equivalence, in particular to reduce the set of contexts one has to consider, see, e.g., [7,3,1,6] and the references therein. For programming languages with references, it is not enough to restrict attention to fewer contexts, since one also needs to be able to reason about equivalence under *related* stores. To address this challenge, methods based on logical relations and bisimulations have been proposed, see, e.g., [8,2,13]. The approaches based on logical relations have so far been restricted to deal only with simple integer references (or references to such). To extend the method to general references in typed languages, one also needs to extend the method to work in the presence of recursive types. The latter is a challenge on its own, since one cannot easily establish the existence of logical relations by induction in the presence of recursive types. Thus a number of research papers have focused on relational reasoning methods for recursive types without references, e.g., [3,1]. Recently, the bisimulation approach has been simplified

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 79–96, 2006.

and extended to work for untyped languages with general references [5,4]. For effectiveness of the reasoning method, we seek *local* reasoning methods, which only require that we consider the accessible part of a store and which works in the presence of a separated (non-interfering) invariant that is preserved by the context. In [2], Benton and Leperchey developed a relational reasoning method for a language with simple references that does allow for local reasoning. Their approach is inspired by related work on separation logic [10,9]. In particular, an important feature of the state relations of Benton and Leperchey is that they depend on only part of the store: that allows us to reason that related states are still related if we update them in parts on which the relation does not depend. In this paper we extend the work of Benton and Leperchey to relational reasoning about contextual equivalence of expressions in a typed programming language with general recursive types *and* general references (thus with higher-order store). We arrive at a useful reasoning method. In particular, we have used it to verify all the examples of [5]. We believe that the method is simple to use, but more work remains to compare the strengths and weaknesses of the method we present here with that of *loc.cit.*

Before giving an overview of the technical development, we now present two examples of pairs of programs that can easily be shown contextually equivalent with the method we develop. The examples are essentially equivalent to (or perhaps slightly more involved than) examples in [5]. Section 5 contains the proofs of contextual equivalence.

The programs M and N shown below both take a function as argument and returns two functions, set and get. In M, there is one hidden reference y, which set can use to store a function. The get function returns the contents of y. The program N uses three local references y_0, y_1 and p. The p reference holds a integer value. The set function updates p and depending on the value of p it stores its argument in either y_0 or y_1. The get function returns the contents of y_0 or y_1, depending on the value of p. Note that the programs store functions in the store. Intuitively, the programs M and N are contextually equivalent because they use *local storage*. The proof method we develop allows us to prove that they are contextually equivalent via local reasoning.

$$
\begin{array}{l}
\mathrm{M} = \mathrm{rec}\ f\ (g\colon \tau \to T\tau')\colon T(((\tau \to T\tau') \to T\mathrm{unit}) \times (\mathrm{unit} \to T(\tau \to T\tau'))) = \\
\quad \mathrm{let}\ y \Leftarrow \mathrm{ref}\ g\ \mathrm{in} \\
\quad \mathrm{let\ set} \Leftarrow \mathrm{val}\ (\mathrm{rec}\ f_{1M}(g_1 : \tau \to T\tau') : T\mathrm{unit} = y := g_1)\ \mathrm{in} \\
\quad \mathrm{let\ get} \Leftarrow \mathrm{val}\ (\mathrm{rec}\ f_{2M}(x : \mathrm{unit}) : T(\tau \to T\tau') = !y)\ \mathrm{in} \\
\qquad (\mathrm{set},\mathrm{get})
\end{array}
$$

$$
\begin{array}{l}
\mathrm{N} = \mathrm{rec}\ f\ (g\colon \tau \to T\tau')\colon T(((\tau \to T\tau') \to T\mathrm{unit}) \times (\mathrm{unit} \to T(\tau \to T\tau'))) = \\
\quad \mathrm{let}\ y_0 \Leftarrow \mathrm{ref}\ g\ \mathrm{in} \\
\quad \mathrm{let}\ y_1 \Leftarrow \mathrm{ref}\ g\ \mathrm{in} \\
\quad \mathrm{let}\ p \Leftarrow \mathrm{ref}\ 0\ \mathrm{in} \\
\quad \mathrm{let\ set} \Leftarrow \mathrm{val}\ (\mathrm{rec}\ f_{1N}(g_1 : \tau \to T\tau') : T\mathrm{unit} = \\
\qquad\qquad\qquad\qquad \mathrm{if\ iszero}(!p)\ \mathrm{then} \\
\qquad\qquad\qquad\qquad\quad (p := 1;\ y_1 := g_1) \\
\qquad\qquad\qquad\qquad \mathrm{else}
\end{array}
$$