Information and Computation 249 (2016) 160–189

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

A Kripke logical relation for effect-based program transformations ☆

Lars Birkedal [a,*], Guilhem Jaber [b,1], Filip Sieczkowski [a,2,*], Jacob Thamsborg [c]

[a] Aarhus University, Denmark
[b] Ecole des Mines de Nantes, France
[c] IT University of Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 2 July 2013
Received in revised form 22 October 2014
Available online 3 May 2016

Keywords:
Effect type system
Kripke logical relation
Program transformation

ABSTRACT

We present a Kripke logical relation for showing the correctness of program transformations based on a region-polymorphic type-and-effect system for an ML-like programming language with higher-order store and dynamic allocation. We also show how to use our model to verify a number of interesting program transformations that rely on effect annotations.

In building the model, we extend earlier work by Benton *et al.* that treated, respectively dynamically allocated first-order references, and higher-order store for global variables. We utilize ideas from region-based memory management, and on Kripke logical relations for higher-order store.

One of the key challenges that we overcome in the construction of the model is treatment of *masking* of regions (conceptually similar to deallocation). Our approach bears similarities to the one used in Ahmed's unary model of a region calculus in her Ph.D. thesis.

1. Introduction

A type system for a programming language classifies programs according to properties that the programs satisfy. An effect system is a type system that, in particular, classifies programs according to which side effects the programs may have. A variety of effect systems have been proposed for higher-order programming languages, e.g., [2–4], see [5] for a recent overview. Effect systems can often be understood as specifying the results of a static analysis, in the sense that it is possible to automatically infer types and effects. Effect systems can be used for different purposes: they were originally proposed by Gifford and Lucassen in [2] as a means of combining functional and imperative features of a language, and for parallelization purposes, but since then they have also been used as the basis for implementing ML using a stack of regions for memory management [4,6] and for region-based memory management in Cyclone, a safe dialect of C [7]. Lately, they have also been used for termination analysis for higher-order concurrent programs [8,9], and for ensuring determinism of parallel

☆ An earlier version of this paper was presented at the ICFP 2011 conference [1]. In addition to including more proofs and examples, the present paper also includes a treatment of an extension of the type system with region polymorphism, which was not covered in [1].

* Corresponding authors. Postal address: Dept. of Comp. Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark.
E-mail addresses: birkedal@cs.au.dk (L. Birkedal), guilhem.jaber@mines-nantes.fr (G. Jaber), filips@cs.au.dk (F. Sieczkowski), thamsborg@itu.dk (J. Thamsborg).
[1] Current affiliation: Université Paris 7, France.
[2] Current affiliation: INRIA Paris, France.

http://dx.doi.org/10.1016/j.ic.2016.04.003
0890-5401/

programs [10,11]. In a recent series of papers, Benton et al. have argued that another important point of effect systems is that they can be used as the basis for effect-based program transformations, in particular compiler optimizations [12–15]. The idea is that certain program transformations are only sound under additional assumptions about which effects program phrases may, or rather may not, have. For example, in a higher-order language with references it is only sound to hoist an expression out of a lambda abstraction if it is known that the expression neither allocates new references, nor reads or writes references.

While it is intuitively clear that effect information is important for validating program transformations, it is surprisingly challenging to develop semantic models that can be used to rigorously justify effect-based transformations. In earlier work, Benton et al. developed semantic relational models of effect systems for a higher-order language with dynamically allocated first-order references (ground store) [14] and for a higher-order language with global variables for higher-order store (but no dynamic allocation) [15].

In this paper we present a Kripke logical relations model of a region-based effect system for a higher-order language with higher-order store *and* dynamic allocation, i.e., with general ML-like references. As pointed out in [15], this is a particularly challenging extension and one that is important for soundness of effect-based transformations for realistic ML-like languages. We now explain what the main challenges are; in Section 3 we give an intuitive overview of how we address these challenges.

The main challenge arises from effect masking: Our region-based effect system includes an effect masking rule that enables hiding local uses of effects. This makes it possible to view a computation as pure even if it uses effects locally and makes the effect system stronger (it can justify more program transformations). To model effect masking we need to model references that, conceptually speaking, become dangling because they point into a region that is masked away. We say "conceptually speaking" because in the operational semantics there is no deallocation of references (the operational semantics is completely standard). However, in our model we have to reason *as if* regions could actually be allocated and deallocated.

Here is a simple concrete example: Consider the following expression e, typed as indicated[3] (where 1 denotes the unit type):

$$\texttt{let x} = \texttt{ref}_\rho\ 7\ \texttt{in let y} = \texttt{ref}_\sigma\ \texttt{x in}$$
$$\lambda z.\texttt{y} := \texttt{x} : 1 \to^{\{wr_\sigma\}} 1, \{al_\sigma\}$$

The program allocates two references, binds them to x and y, and then returns a trivial function from unit to unit that assigns x to y. The types indicate that x will be bound to a location in region ρ, say location 0, and that y will be bound to a location in region σ, say location 1. At location 0 the value 7 is stored and at location 1 the location 0 is stored. The so-called latent effect $\{wr_\sigma\}$ of the function type of the whole expression describes which effects the function may have when called. Finally the entire expression has the effect $\{al_\sigma\}$ as it allocates a location in region σ. It performs allocation in region ρ as well, but this effect has been masked out: since no ρ appears in the return type, the expression cannot leak location 0, and we can, conceptually, deallocate all locations in region ρ once the computation has run; this is what the masking rule captures. This particular example, however, was chosen to stress test the masking rule, as location 0 is in fact leaked: it is in the function closure that the expression returns. The function neither reads not writes location 0, but it does write it to the heap; our model must be able to cope with such conceptually dangling pointers. On the level of types, the meaning of the type $\mathsf{ref}_\rho\mathsf{int}$ of x changes: after the allocation of values for x and y, it contains location 0, but what should it contain after region ρ has been masked out? If, e.g., $\mathsf{ref}_\rho\mathsf{int}$ is taken to be empty, then the function is most likely no longer well-typed. We explain our answer to this question in Section 3. Since we have to reason as if regions are deallocated, it is, in hindsight, not so surprising that our approach is closely related to the one used by Ahmed in her unary model of a region calculus with region deallocation [16].

One could argue that the program above should in fact be treated as semantically pure, even if the type system cannot infer this fact, since the location bound to y is never made accessible to the context. Thus, one might want to provide a stronger masking principle on the semantic side. We do not pursue that goal in the current paper and hence our semantic treatment of masking mimics the syntactic masking rule. In future work, it will be interesting to investigate a more liberal notion of semantic masking.

Note that the effect annotations in the types are just annotations; the operational semantics is completely standard and regions only exist in our semantic model, not in the operational semantics. Further note that the issues in the above example do not arise for a language with only ground store.

Another challenge arises from the fact that since our language includes dynamically allocated general references, the existence of the logical relation is non-trivial; in particular, the set of worlds must be recursively defined. Here we define the worlds as a solution to a recursive metric-space equation, building on our earlier work [17], which gives a unified account of models based on step-indexing [16] and on domain theory.

Yet another challenge arises from the fact that our language includes a new form of region polymorphism, inspired by [3], but not covered before in semantic models for validating effect-based program equivalences. Region polymorphism makes

[3] Note that conceptually the whole expression could be considered pure: none of its effects can actually be observed by any context. However, our model and type-and-effect system are not fine-grained enough to express this.