A calculus for logical refinements in separation logic

Dan Frumin
ICIS, Radboud University
dfrumin@cs.ru.nl

Robbert Krebbers
Delft University of Technology
mail@robbertkrebbers.nl

Abstract

We present a formalization of logical refinements for programs in a higher-order programming languages with general references, polymorphism and existential types, and concurrency. The logical refinement is sound w.r.t. contextual refinements of programs, and has been formalized in Coq, together with a number of case studies, in state of the art concurrent separation logic Iris. The work extends on the initial interpretation of logical relations in Iris by Krebbers *et al.*, to the extent that we provide a calculus that allows one to reason about logical refinement in an abstract way using symbolic execution, together with a collection of Coq tactics that make reasoning in the calculus tractable for realistic examples.

1 Introduction

Reasoning about equivalence of programs is an old problem in semantics of programming languages, with applications many applications including program verification and compilation. One of the most widely used notions is that of *contextual refinement*. Intuitively, e_1 contextually refines e_2 if all the observable behaviours of e_1 can be observed by e_2 as well. However, proving contextual refinement of two given programs is tricky, as it involves considering *all the possible* program contexts. One of the techniques proposed to resolve this are *logical relations*.

The aim of this work is to describe a system for formal reasoning about contextual refinements through logical relations for System F with state, existential types, and concurrency primitives. The system is built on top of the powerful higher-order separation logic *Iris* [4]. This allows the user to leverage advanced features of Iris, such as ghost state and invariants [3]. Furthermore, Iris is implemented on top of Coq [6], ensuring that the only trusted base for our development is the Coq kernel itself.

This work is based on the earlier implementation of logical relations in Iris using the "Interactive Proof Mode" by Amin Timany, Robbert Krebbers, and Lars Birkedal [6]. In that work, all the refinements were proved by explicit reasoning in the model of logical relations, i.e. by unfolding the Iris definition of logical relation. Our work scales up the development by

- providing a calculus for reasoning about logical relatedness in a self-contained abstract way for a large class of logical refinements using symbolic execution and compatibility lemmas;
- extending the definition of logical relations to *masked logical relations* to reason abstractly about invariants;
- providing facilities in Coq such as tactics and tactic lemmas for showing refinements of concrete programs;
- providing support for existential types, for the purposes of showing representation independence theorems.

As a result of those changes, the compilation times for the examples improved and the proofs became cleaner. The full source code containing the whole development and examples can be found at https://gitlab.mpi-sws.org/dfrumin/logrel-conc.

CoqPL, January 13, Los Angeles, California, United States
2018.

$$
\begin{aligned}
e ::= {} & () \mid l \in \mathit{Loc} \mid n \in \mathbb{N} \mid b \in \mathbb{B} \mid \mathsf{rec}\ f\ x := e_1 \\
& \mid (e_1, e_2) \mid \mathsf{inl}\ e_1 \mid \mathsf{inr}\ e_1 \mid x \in \mathit{Var} \\
& \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
& \mid \pi_1\ e_1 \mid \pi_2\ e_1 \mid \mathsf{case}(e_1, e_2, e_3) \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \\
& \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ x\ \mathsf{in}\ e_1 e_2 \mid \mathsf{fork}\{e\} \mid \mathbf{new}\ e \\
& \mid !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \ldots \\
\tau ::= {} & \alpha \mid \mathbf{N} \mid \mathbf{2} \mid \tau \times \tau \mid \tau \to \tau \mid \tau + \tau \mid \mathbf{ref}\ \tau \mid \forall \alpha.\tau \mid \exists \alpha.\tau \mid \mu \alpha.\tau
\end{aligned}
$$

Figure 1. Syntax of the object language

2 Basic judgements and rules

The syntax of the language at hand is presented in Figure 1. The operational semantics is given using a standard interleaving semantics. The typing rules are fairly standard and omitted.

The simplified basic judgments of the calculus we are considering are of the form

$$\Delta; \Gamma \models e_1 \lesssim e_2 : \tau \tag{1}$$

which states that

1. two (possibly open) expressions e_1 and e_2 are logically related at a (possibly open) type τ;
2. under the assignment Γ of types to free term variables, and under the assignment Δ of relations to free type variables;

The idea of the refinement is that e_2 *simulates* e_1: for each execution step of e_1, the expression e_2 may be executed for several steps. This guarantees that e_1 is not going to contain any observable behaviour that e_2 cannot exhibit.

There are two main ways of deriving judgements like in Eq. 1: compatibility lemmas, and symbolic execution rules.

Compatibility lemmas. Compatibility lemmas essentially say that the logical relation is closed under typing-like rules. For instance, the compatibility rule for pairs state that

$$\frac{\Delta; \Gamma \models e_1 \lesssim e_2 : \tau \qquad \Delta; \Gamma \models e_1' \lesssim e_2' : \sigma}{\Delta; \Gamma \models (e_1, e_1') \lesssim (e_2, e_2') : \tau \times \sigma}$$

And the compatibility lemma for modules say that

$$\frac{\Delta\,[\alpha \leftarrow R]; \Gamma \models e \lesssim e' : \tau \qquad R \in \mathcal{P}(\mathit{Val} \times \mathit{Val})}{\Delta; \Gamma \models \mathsf{pack}\ e \lesssim \mathsf{pack}\ e' : \exists \alpha.\tau}$$

Using all the compatibility lemmas we can prove the fundamental property of logical relations.

Theorem 2.1. *If* $\Gamma \vdash e : \tau$*, then for all* Δ *we have* $\Delta; \Gamma \models e \lesssim e : \tau$.

Symbolic execution rules. Symbolic execution rules allow us to prove refinements of programs that are actually syntactically different. Programs on the right and left hand sides operate on different symbolic heaps: the standard "points to" assertion from separation logic is denoted as $l \mapsto_s v$ for the right hand side, and as $l \mapsto_i v$

for the left hand side. The basic symbolic execution rule for pure reductions (*i.e.,* reductions that do not have effects modifying the state or the thread pool, *e.g.,* beta reduction) is the following:

$$\frac{\triangleright(\Delta;\Gamma \models K[e'] \lesssim t : \tau) \qquad \mathrm{pure}(e, e')}{\Delta;\Gamma \models K[e] \lesssim t : \tau}$$

Note that the pure reduction rule for the left hand side gives us a *later modality* $\triangleright$ for the new goal, which allows us reason about recursive programs using Löb induction. There is a similar rule for the right hand side:

$$\frac{\Delta;\Gamma \models t \lesssim K[e'] : \tau \qquad \mathrm{pure}(e, e')}{\Delta;\Gamma \models t \lesssim K[e] : \tau}$$

The rules for stateful reductions on the right hand side are akin to the symbolic execution WP-rules in Iris [4, 5], but adapted for the binary setting. For instance, below is the rule for symbolically executing *store* operation on the right hand side.

$$\frac{(l \mapsto_s -) \ast (l \mapsto_s v \mathrel{-\!\!*} \Delta;\Gamma \models t \lesssim K[()] : \tau)}{\Delta;\Gamma \models t \lesssim K[l \leftarrow v] : \tau}$$

Coq formalization. Such rule is formalized in Coq as follows:

```
Lemma bin_log_related_store_r Δ Γ K l e v v' t τ :
  to_val e = Some v' →
  l ↦ₛ v -∗
  (l ↦ₛ v' -∗ {Δ;Γ} ⊨ t ≤log≤ fill K (#()) : τ) -∗
  {Δ;Γ} ⊨ t ≤log≤ fill K (#l ← e) : τ.
```

The rule is then viewed as a theorem inside the logic Iris. In addition to such a theorem each corresponding rule has Coq tactic associated with it that allows for the automatic discharging and introduction of premises (such as $l \mapsto_s v$ and $l \mapsto_s v'$) when used in the Interactive Proof Mode [6].

Example scenario. Suppose we wish to prove the following theorem:

```
Lemma dummy Δ Γ l l' :
  l ↦ᵢ #2 -∗ l' ↦ₛ #0 -∗
  {Δ;Γ} ⊨ !#l ≤log≤ (#l' ← #2;; !#l') : TNat.
```

It states that, under the assumption that the locations `l` and `l'` have values 1 and 0, respectively, the program on the left hand side refines the program on the right hand side. After introducing the assumptions, the user's Coq goal becomes:

```
Δ : list D
Γ : stringmap type
l, l' : loc
_____________________________________
"Hl" : l ↦ᵢ #2
"Hl'" : l' ↦ₛ #0
-------------------------------------∗
{Δ; Γ} ⊨ ! #l ≤log≤ (#l' ← #2;; ! #l') : TNat
```

At this point the user can invoke the tactic `rel_store_r`, which uses the theorem `bin_log_related_store_r` to transform the goal into:

```
"Hl" : l ↦ᵢ #2
"Hl'" : l' ↦ₛ #2
-------------------------------------∗
{Δ; Γ} ⊨ ! #l ≤log≤ (#();; ! #l') : TNat
```

After using the symbolic execution tactics `rel_pure_r`, `rel_load_r` and `rel_load_l` the goal further reduces to the statement

```
"Hl" : l ↦ᵢ #2
"Hl'" : l' ↦ₛ #2
-------------------------------------∗
{Δ; Γ} ⊨ #2 ≤log≤ #2 : TNat
```

which is an instance of the compatibility lemma for natural numbers.

3 Implementation and case studies

All the material mentioned in this abstract has been formalized in Coq, including the new masked interpretation of logical relatedness, all the rules, and the soundness of the system as a whole. The conciseness and compilation time of the formalization has benefited from the changes outlined in the introduction, as well as from switching from De Bruijn indices to explicit named representation.

In addition to the system itself, we have formalized several examples entirely inside the calculus:

1. refinements of coarse-grained concurrent data structures by fine-grained ones: concurrent counter and Treiber stack;
2. the ticket-based lock refinement of the spin lock;
3. generative ADT example from [1];
4. algebraic laws for the non-determinism operator defined via concurrency
5. various examples of higher-order functions with state from [2] (those that still hold in the presence of concurrency).

The first two refinements were already present in the previous version of the library [6], but they have greatly benefited from the changes that we have introduced. Aside from better readability and better decomposition of the proof, the compilation time has dropped significantly. For the Treiber stack example it is 1 minutes 17 seconds in the new version versus 2 minutes 45 seconds in the old version[1].

4 Future work

Possible future work directions include enriching the type system and the object language, and increasing expressivity for formalizing more complex examples, such as refinements of programs involving speculation or helping and side-channels.

We have managed to formalize a refinement of a coarse-grained concurrent stack by a stack with helping; however, that proof requires us to appeal to the interpretation of the logical relation judgements and it seems that it is not currently possible to formalize such an example in its entirety without reasoning explicitly in the model, although parts of the proof are still carried out in the calculus.

References

[1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent Representation Independence. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1480881.1480925

[2] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* 22, 4-5 (2012), 477–528.

[3] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

[1] The compilation times were measured several times and averaged on a two-core 2.50GHz processor running Coq 8.6 under 8Gb of RAM.

[4] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Submitted for publication* (2017).

[5] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag New York, Inc., New York, NY, USA, 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[6] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. *SIGPLAN Not.* 52, 1 (Jan. 2017), 205–217. https://doi.org/10.1145/3093333.3009855