Predoc Internship

Introductory report

Toward a formalization in Coq of a parallelization theorem for a concurrent language with general references: an introduction to logical relations

Author:
Yannick Zakowski

Supervisor:
Lars Birkedal
Logics and Semantics group

Abstract

Birkedal *et al.* have recently designed a logical relation for showing the correctness of program transformations for a concurrent language with higher-order functions as well as dynamic memory allocation. In particular a Parallelization Theorem is proved, expressing when running two expressions in parallel rather than sequentially is sound.

However, such a result requires to be able to prove contextual equivalence over a language with rich features. Performing so straightforwardly does not seem to be realistic, hence the need for a tool. While several have been proposed, the one of use here is the notion of a logical relation based on a semantic interpretation of types. We aim through this report at motivating and introducing this idea over increasingly complex languages.

A precise understanding of every layers involved is indeed required, the objective of the stage itself being the formalization in Coq of the Parallelization Theorem.

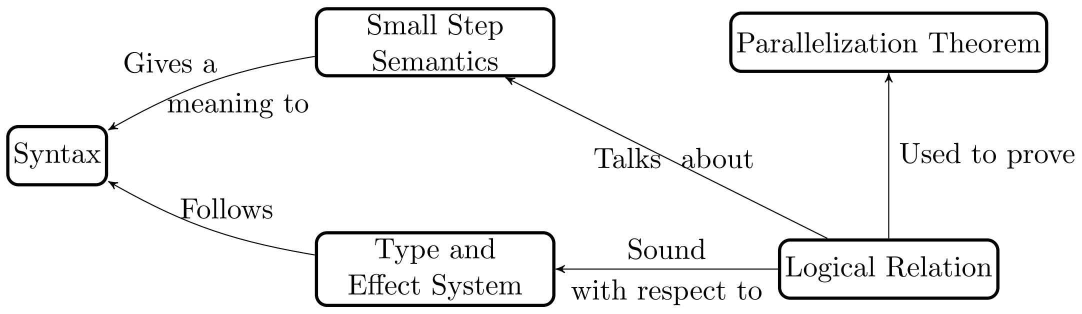

Figure 1: Schematic display of the components leading to the Parallelization Theorem.

1 Introduction

Relational reasoning about program equivalence allows one to reason about the correctness of program transformations, such as an optimization that a compiler could perform. The relevant notion of equivalence here is the one of contextual equivalence: the semantic equivalence is asserted in any possible context. Improvement of reasoning methods for higher-order languages with general references has been the focus of numerous recent research, building upon different tools such as bisimulations, traces, game semantics or the one we are focus on, logical relations. Following this tradition, Birkedal *et al.* recently managed to design and prove a so-called Parallelization Theorem for a rich concurrent language [6]: impredicative recursive types, recursive functions and general ML-like references. Such a groundbreaking result means that they are able to specify a static analysis whose result can safely allow to decide whether two sequential expressions can rather be executed in parallel without modifying their semantics.

Figure 1 displays the various main components we are interested in. The language is defined through a syntax, whose meaning is provided through a semantics, considered operational for our purpose here. The specification of the static analysis is given under the form of an enriched type system, collecting typing information as well as over-approximating the memory accesses. The Parallelization Theorem states that given a condition over the result of the static analysis - the effects obtained by typing both expressions "do not interfere" -, we obtain a property of the semantics - both the programs $e_1;\ e_2$ and $e_1 \mid\mid e_2$ have the same semantics -.

However, a fifth component must be highlighted. Indeed, by nature, a type system is tightly bound to the syntax, while we try to prove a property involving subtle reasoning about the semantics. More precisely, we are trying to show a contextual equivalence: the semantics must be guaranteed preserved whatever context these expressions are executed in. Additionally, recursive types and references can be instantiated with types or values of

$$\begin{array}{rcl}
\tau & ::= & \mathit{unit} \mid \mathit{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\ \alpha \cdot \tau \mid \Lambda\ \alpha \cdot \tau \mid \mathit{ref}\ \tau \\
v & ::= & \mathit{unit} \mid x \mid (v_1,\ v_2) \mid \lambda \cdot e \\
e & ::= & v \mid \mathit{proj}_i\ v \mid v\ e \mid \mathit{ref}\ v \mid !v \mid v_1 := v_2 \mid (v_1 \mid\mid v_2) \\
E & ::= & [] \mid v\ E \mid (E \mid\mid v) \mid (v \mid\mid E)
\end{array}$$

Figure 2: Syntax of our language

any structural size. Therefore, it is very hard to work directly over a syntactically based approach to prove such a result: we need an auxiliary tool. One solution is to build a logical relation upon a semantic interpretation of types: we build a relation which follows the syntax, as the type system does, but directly talks about the semantics.

Nevertheless, building such a logical relation is quite a task in itself. While the formalization of the Parallelization Theorem and its proof is the objective of the internship, we will hence focus in this intermediate report on motivating and constructing a logical relation for a realistic language. More precisely, we come back to a syntactic approach in order to motivate the need for a semantic way of reasoning in Section 2. We then sketch the incremental building of the logical relation for progressively richer languages: section 3 restricts itself to System F, while section 4 and 5 respectively add recursive types and references. We in particular present in deeper details the logical relation for this language, before sketching in Section 6 the missing ideas to reach the logical relation designed by Birkedal *et al.* for a concurrent **ML**-like language [6].

The syntax of the targeted language is described in Figure 2. However, we will work through this report on various subsets of this language in order to gradually introduce the concepts. We assume a completely standard type system and operational semantics over it.

We precise that the object of the internship is the formalization in Coq of the Parallelization Theorem. However, we introduce through this midterm report the underlying techniques at stake. Those techniques have been developed through numerous researches during the last decades, and notably extended and refined to realistic languages by Birkedal *et al.* during the past ten years. We do not focus on our personal production here.

2 A syntactic proof of soundness

While the syntactic approach will not be the one followed, its study is of interest in order to understand its inadequacy. Schematically, such a proof is shaped according to the following pattern: given a language, we grant it with both a type system and a semantics, and prove a property relating them. The most famous example has been brilliantly advertised by Robin Milner in 1978:

> Well-typed programs cannot go "wrong".

Intuitively, this states that any well-typed expression actually reduces to a value having the same type. Under more formal terms, this can be phrased as such:

Theorem 1. *Soundness. If* $\Gamma \vdash t : \tau$ *and* $t \to^* t'$ *and* t' *is irreducible, then* $t' \in val$ *and* $\Gamma \vdash t' : \tau$.

However, we are more interested here in the proof of the theorem, built upon two major lemmas, than in the theorem itself. The first lemma, *preservation*, essentially states that the type judgment is invariant under the semantics.

Lemma 1. *Preservation. If* $\Gamma \vdash t : \tau$ *and* $t \to t'$, *then* $\Gamma \vdash t' : \tau$.

The second one, *progress*, expresses that any well-typed, closed expression is either a value, or can take one step of reduction.

Lemma 2. *Progress. If* $\Gamma \vdash t : \tau$, *then* t *is a value, or there exists* t' *such that* $t \to t'$.

Three remarks are of interest here.

First, we would naturally like to only have to type terms which actually can be written in our language. However, sticking strictly to the syntax, this cannot be sufficient when proving the preservation lemma by case study over the step reduction. Indeed, think of a language with references: one cannot write a term containing an explicit location, nevertheless a term containing a reference will reduce to one such. Hence the need for a typing rule for locations, which rises the need for enhancing the typing judgment with heap contexts. We would like to avoid our type system to grow more complicated than needed.

Now, note that in the desired Parallelization Theorem, we try to prove a property relating two terms together. We hardly see how a purely syntactic argument could allow us to perform so. What we would like intuitively is to prove that they both belong to some sort of set of expressions which share common properties. The notion of semantic interpretation and logical relation that we will present here is a tool which can allow in particular such a relational reasoning.

A last illustration of the inadequacy of purely syntactic proofs, hence a motivation for a new tool for reasoning, already rises in a unary setup, with a language as simple as the simply typed lambda-calculus. As exposed in Chapter 12 of Pierce's book [9], such a restricted language satisfies quite a strong property: every well-typed term is normalizable, *i.e.* every well-typed program is guaranteed to halt in a finite number of steps. However, if we were to try to prove this result by following the syntax, going by induction on the size of a well-typed term, we would not be able to cope with the application case. Indeed, knowing that both t_1 and t_2 are normalizing respectively to v_1 and v_2, we try to show that $t_1\ t_2$ is normalizing. Deducing from the typing judgment that $v_1 = \lambda x.t'_1$, we are left with proving that $t'_1[x \mapsto v_1]$ is normalizing. But this term can be bigger than the original, hence an impossibility to conclude. Actually, what we need is a stronger induction principle than the purely syntactic one. In order to get one such, a trick is to interpret types as predicates over terms in order to define a logical relation.

3 A logical relation for Girard's system F

As we have seen, reasoning directly on the syntax of a term, or through its typing judgment, does not provide a strong enough induction principle for many applications. One way to ease such reasoning is the notion of logical relation: a relation which relies on the semantics, but respect the syntax in the sense that they are compatible with each typing rule. In order to build one such relation, we want to interpret each syntactic type of the language as a so-called semantic type, taking its values over a universe T.

In order to progressively illustrate this technique, we start with Girard's system F. For such a restricted language, T will quite naturally be a set: to each type we associate a predicate over values, characterizing values which have this type. The definition hence is by induction on the syntactic type τ as follow:

$$\begin{aligned}
[\![int]\!] &= \{n \mid n \in \mathbb{N}\} \\
[\![\tau_1 \to \tau_2]\!] &= \{\lambda x.e \mid \forall v_1 \in [\![\tau_1]\!],\ e[x \mapsto v_1] \in \mathcal{E}[\![\tau_2]\!]\} \\
[\![\forall \alpha.\tau]\!] &= \bigcap_{R \in T} ([\![\tau[\alpha \mapsto R]]\!])
\end{aligned}$$

where $\mathcal{E}[\![\bullet]\!]$ is the extension of the interpretation of types over expressions, intuitively defining the expressions which reduce to a value in the interpretation[1]:

$$\mathcal{E}[\![\tau]\!] = \{e \mid \forall v\ \textit{value} \text{ such that } e \to^* v,\ v \in [\![\tau]\!]\}$$

Having built such an interpretation, the logical relation is defined straightforwardly: a term t is logically related to a type τ if t belongs to the interpretation of τ. To illustrate how such a process can be of use, let us come

[1]One could wonder if this definition is not circular. However, the extension to expressions can be defined independently for any predicates over values, and instantiated with the interpretation of τ in the interpretation of the arrow-type.

back to the proof of normalization for the simply typed calculus. Using our logical relation as a middle-man now allows us to prove the normalization lemma in two steps: first that any term logically related to a type is normalizable, second that any well typed term of type τ is logically related to τ. The first step is trivial by induction on the syntax of τ. The second property is usually called the fundamental theorem of logical relations, thanks to which it deserves its name. The standard proof method is to go on by induction over the type judgment, deducing it from so-called compatibility lemmas asserting that each typing rule remains sound when substituting the typing judgment by the logical relation.

This simple example illustrates how the fundamental theorem of logical relations allows us to reason about our language from a semantical point of view while proving the correction of syntactically driven analyses. We present in the remaining of this report how to enrich the model in order to deal with richer languages.

4 Extending to recursive types: the need for a metric structure

Everything hence seems to be quite straightforward for system F. But what if we want to extend our language with impredicative recursive types? Intuitively, we would like to define their interpretation as something along those terms:

$$[\![\mu\alpha.\tau]\!] = fix(fun\ R \mapsto [\![\tau[\alpha \mapsto R]]\!])$$

But for this definition to have a sense, we cannot anymore simply take T to be any set. We need to have some sort of metric over it, so that we can enforce the functor $fun\ R \mapsto [\![\tau[\alpha \mapsto R]]\!]$ to be contractive, hence ensuring the existence and unicity of the fix-point.

Several solutions to this problem have been proposed, such as Amadio's recursion over realizability structures [3], or the step-indexing models introduced by Appel and McAllester [4, 2]. The rough intuition behind step-indexing is to stop considering values being of a certain type, but values behaving as being of a certain types up to a certain amount of computational steps. To illustrate it, consider the type $ref(ref\ int)$. If you are only interested in one step of computation, a reference over any value will behave as if being of this type. Up to two step of computation, the interpretation is a subset of the previous one: the value have to be a reference over a reference. With three step or any lesser degree of approximation, the interpretation will be precisely the references over references over values of type int. We then say that a value truly belong to the interpretation of a type if it belongs to this interpretation for any level of approximation, *i.e.* any number of computational steps.

In order to implement this intuition, our semantic types are not anymore predicates over sets, but must be indexed by the level of approximation we work on. Additionally, this indexation must naturally be downward closed: we therefore work on so-called uniform predicates over values. This new setup allows us to actually decrease the size of our types at each interpretation with respect to a lexicographic order over step-indexed first, structure of terms second.

5 Extending to general references: a Kripke-like parametrization

Getting closer to a realistic language, we now want to extend it with **ML**-like general references. However, this addition entails a quite major shift: the interpretation of a type $ref\tau$ must depend on the current state of the memory. In order to cope with this, we adopt a model of possible worlds, so-called Kripke worlds by analogy with the modal logic.

The interpretation of a type is therefore not anymore directly a predicate, but must now be parametrized by a world, whose aim is to describe the memory. What hence shall be this world? We want to associate to each location a semantic type so that roughly speaking, we obtain:

$$[\![ref\tau]\!]\ w = \{l \mid w\ l = [\![\tau]\!]\}$$

Hence, if we sum up and make things a bit more precise, we need:

$$\begin{aligned} T = W \rightarrow_{mon} UPred(Val) \\ W = Loc \rightharpoonup T \end{aligned}$$

Which amount to quite an issue if we unfold our definition: the equation $T = Loc \rightharpoonup T \rightarrow_{mon} UPred(Val)$ admits, for cardinality reasons, no solution in Set. This constatation led to several propositions to solve approximate versions of those. However, Birkedal *et al.* managed to show that this equation admits a solution over an appropriate simple category of metric space: the category of complete 1-bounded ultrametric spaces, denoted $CBUlt_{ne}$.

5.1 Solving the equation over $CBUlt_{ne}$

We won't go here in the mathematical details, we simply review a few basic facts of use for our purpose. An ultrametric space is a metric space whose distance d satisfies a stronger triangle inequality, namely: $\forall x\ y\ z,\ d(x, z) \leq max(d(x, y),\ d(y, z))$. One such space is said to be complete if every Cauchy sequence admits a limit, and 1-bounded if its distance function takes values in $[0, 1]$. A function $f\ :\ (X_1, d_1) \rightarrow (X_2, d_2)$ is said to be non-expansive if $\forall x\ y \in X_1,\ d_2(f(x), f(y)) \leq d_1(x, y)$ and contractive if $d_2(f(x), f(y)) \leq \delta \cdot d_1(x, y)$ for some $\delta < 1$.

Taking the complete, 1-bounded, non-empty, ultrametric spaces as objects and the non-expansive functions between them as arrows grants us with a category named $CBUlt_{ne}$. This category have enough good properties to satisfy a result similar to the well-known Banach theorem in a more conventional context:

Theorem 2. *Any locally contractive functor over $CBUlt_{ne}$ admits a unique (up to isomorphism) fix-point.*

5.2 A general recipe

Following the previous result, one can obtain such a general recipe:

- Choose a domain V for semantics values.
- Define an object $Pred(V)$ in $CBUlt_{ne}$ to represent predicates over values.
- Solve the following recursive domain equation in $CBUlt_{ne}$:

$$\hat{T} = \frac{1}{2} \cdot ((Loc \rightharpoonup_{fin} \hat{T}) \rightarrow_{mon} Pred(V))$$

- Define the desired domains of semantics types and worlds as so:

$$W = Loc \rightharpoonup_{fin} \hat{T}, \qquad T = W \rightarrow_{mon} Pred(V)$$

The monotonicity in the definition of semantic words is defined with respect to the extension order over worlds $w \sqsubseteq w'$ stating that the domain of w' contains the domain of w, and that both world agrees on it. It is interesting to have an intuition about the interest of this enforcement.

It is interesting to observe how generic is this recipe, allowing for different models depending of your choice of V and $Pred(V)$. A first use of this method has been made by Birkedal *et al.* by constructing V and $Pred(V)$ using domain theory [7]. This approach arguably presents two drawbacks: it requires some non-trivial domain theory, and more fundamentally is tied to denotational semantics. While this last point is in general not an issue when reasoning in term of contextual equivalence, this claim is no longer true when in presence of concurrency. Indeed, one must be able to reason for any possible interleaving, which inherently calls for a reasoning tied to the operational semantics.

5.3 A logical relation built upon a step-indexed model

As we stated, the recipe is general, and Birkedal *et al.* also cooked using operational semantics [5]. In order to do so, the parameter V, modeling our values, is taken as the set of closed syntactic values from the operational semantics. The predicates over values are, as motivated through the previous section, modeled as predicates on step-approximated values, *i.e.* subsets of $\mathbb{N} \times V$ that are downwards closed in the first component. More formally, we work over the so-called uniform predicate on V:

$$UPred(V) = \{P \subseteq \mathbb{N} \times V \mid \forall (k, v) \in P,\ \forall j \leq k,\ (j, v) \in P\}$$

As explained in Section 4, we motivate the use of uniform predicates as a way to enforce a structural decrement that is no longer directly enforced on the syntax as soon as our language includes recursive types. Intuitively, we interpret types approximatively up to a level k by gathering values which can't be denied to be of this type in less than k steps of computation. A value is then in the interpretation of a type if it so up to any approximation. While this idea of a step-indexed model is not recent, having been introduced by Appel and McAllester in 2001 [4], the idea actually fits in this new framework. Indeed, $UPred(V)$ can be made an instance of $CBUlt_{ne}$: in order to do so, we build the distance which intuitively decreases with the amount of steps for which two values are indistinguishable, asymptotically being at distance 0 if they coincide. More formally, defining for any $P \in UPred(V)$ its *k-th* approximation as $P_k = \{(m, v) \in P \mid m \leq k\}$, we supply the space with the following distance:

$$d(P, Q) = \begin{cases} 2^{-max\{k \mid P_k = Q_k\}} & if P \neq Q \\ 0 & otherwise \end{cases}$$

Equipped like so, $UPred(V)$ can be proved to be an object in $CBUlt_{ne}$, hence granting us with two domains W of worlds and T of semantic types on which we can now work to model our language.

In order to do so, as previously stated, we define the semantic interpretation of types, associating to each syntactic type a semantic one. Unfolding the definitions, we recall that a semantic type is a monotonic function from worlds, modeling the memory, to uniform predicate over V, modeling values. The detail of the interpretation is given in Figure 3. Note that we only consider *int* as a base type for convenience of presentation, but could equivalently consider any set of base types.

The interpretation of the *unit* and *int* types are straightforward: the only values which acts as being of one of those types for at least one step of computation are the ones which have exactly one of those types, *i.e.* respectively the *unit* value or an *int* value. A product type is interpreted straightforwardly by induction: a product value is approximately of one

$$
\begin{aligned}
[\![\tau]\!] &: W \to_{mon} UPred(Val) \\
[\![1]\!]\ w &= \{(k,\ unit) \mid k \in \mathbb{N}\} \\
[\![int]\!]\ w &= \{(k,\ n) \mid k, n \in \mathbb{N} \\
[\![\tau_1 \times \tau_2]\!]\ w &= \{(k, (v_1 \times v_2) \mid (k,\ v_i) \in [\![\tau_i]\!]\ i = 1, 2\} \\
[\![ref\ \tau]\!]\ w &= \{(k,\ l) \mid l \in dom\ w \wedge w(l) =_k [\![\tau]\!]\} \\
[\![\tau_1 \to \tau_2]\!]\ w &= \{(k,\ v) \mid \forall w' \sqsupseteq w,\ \forall\ i \leq k, \\
&\qquad (i,\ v') \in [\![\tau]\!]\ w' \Rightarrow (i,\ v\ v') \in \mathcal{E}[\![\tau']\!]\ w'\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\tau]\!]\ w = \ & \{(k,\ t) \mid \forall j \leq k,\ \forall h, h', e', \\
& (h :_j w \wedge (t|h) \to^j (e'|h') \wedge (e', h') \text{ irreducible}) \Rightarrow \\
& (\exists w' \sqsupseteq w,\ h' :_{k-j} w' \wedge (k-j, e') \in [\![\tau]\!]\ w')\}
\end{aligned}
$$

$$
h :_k w \Leftrightarrow dom(h) = dom(w) \wedge \forall i \leq k,\ \forall l \in dom(w),\ (i, h(l)) \in w(l)(w)
$$

Figure 3: Semantic interpretation of types for the language $F_{\mu,\ ref}$

$$
\begin{aligned}
[\![\Gamma]\!] &: W \to UPred(V^{\Gamma}) \\
[\![\emptyset]\!]\ w &= \{(k,\ \emptyset) \mid k \in \mathbb{N}\} \\
[\![(x, \tau), \Gamma]\!]\ w &= \{(k,\ \rho[x \mapsto v]) \mid (k,\ \rho) \in [\![\Gamma]\!]\ w \wedge (k,\ v) \in [\![\tau]\!]\ w\}
\end{aligned}
$$

$$
\Gamma \vDash t : \tau \Leftrightarrow \forall k \in \mathbb{N},\ \forall w \in W,\ \forall \rho,\ (k,\ \rho) \in [\![\Gamma]\!]\ w \Rightarrow ((k,\ \rho(t)) \in \mathcal{E}[\![\tau]\!]\ w\}
$$

Figure 4: Interpretation of contexts and logical relation

such type if it is true with the same degree of approximation componentwise. The reference type is more interesting: as explained in the beginning of this Section, the interpretation depends on the current state of memory, which we have modeled through words. A reference on a type τ hence is interpreted as the set of locations which are mapped in the current world to the interpretation of τ, up to the desired degree of approximation.

The arrow type finally is interpreted the same way it already was for , using an extension to the interpretation of types over expressions, but the setup got a bit more complicated. First, we now have to relate our high level model of the memory, the worlds, with the low level one used in the semantics, the heap. We do so through a predicate $h :_k w$ stating that h and w have the same domain, and that at any location l in their domain, h points to a value which is in the semantic type w points to at this location, up to approximation k. The second point worth noticing is the introduction of a futur world w', extending w. This is crucial to take into account the intuitive idea that lambda abstraction suspends computation, the state of the memory might have changed between the evaluation of the lambda abstraction and the one of its argument. The condition of monotonicity we put in the very definition itself of the notion of semantic type is here crucial: new allocations or updates can have occurred, but any location previously allocated will still point to the same type.

Note that this interpretation of types require a few sanity checks. Each inductive case must be proved to be non-expansive, monotonic, and actually take its value over the uniform predicates on V.

As detailed in Figure 4, one last step is required before going on defining the logical relation: since we want to be able to reason about non closed terms, we need to interpret the variable context. The definition is straightforward as a world-indexed uniform predicate over mapping from program variables to values.

Finally, the logical relation is a judgment similar to the typing one, having the form $\Gamma \vDash t : \tau$. It states that for any degree of approximation k, for any world w, for any substitution ρ in the interpretation of Γ, the closed expression resulting from applying ρ to t is in the interpretation of the type τ in the world w up to approximation k.

This definition of logical relation allows us to prove the so-called Fundamental Theorem of Logical Relations: it actually is a logical relation in the sense that it follows the type system.

Theorem 3. *Fundamental Theorem of Logical Relations. If $\Gamma \vdash t : \tau$ then $\Gamma \vDash t : \tau$.*

This result is the direct consequence of compatibility lemmas: for each type judgment, we prove the adequacy of the logical relation with the judgment. Note that we are able to prove this statement with a typing system which only types terms we actually can write in the language. Being on the other hand able to prove that "logically related programs cannot go wrong", we obtain a semantic type soundness result, similar to the syntactic one, but with a simpler type system. However, as argued when motivating the use of logical relations, what we look for is actually a relationship between expressions, allowing us to prove a notion of contextual equivalence. Precisely, the work we presented here in a unary context for sake of clarity is by no mean tied to it, we can do the same by working over uniform relationships instead of uniform predicates. We obtain by doing so a logical relation between expressions which actually entails the contextual equivalence.

However, this logical relation is still limited in practice. Indeed, worlds describe invariants on heap. With such a simple model, these invariants only state that two locations store values of related type, which is often insuffisant: we need richer worlds.

6 Toward the Parallelization Theorem and even further: regions and richer worlds

At this point of the internship, the theoretical study coming along with the **Coq** formalization, remaining extensions are not mastered. We however briefly sketch them here as a road map to come for the incoming months.

The first point to recall is that now that we deal with a concurrency, we do not restrict ourselves to type systems. Indeed, two prove that two expressions can safely be parallelized, we crucially need to restrict the set of possible context in which ones we check contextual equivalence. For instance, considering as in [6] the two expressions

$$e_1 \equiv x := 1;\ y := 1 \text{ and } e_2 \equiv y := 1;\ x := 1,$$

we want to be able to assert that no other threads, *i.e.* no part of the context, can have access to x or y and modify them. In order to express such a restriction, the type system is refined in a type-and-effect system [10] over-approximating the different memory accesses realized: writes, read and allocations. In order to model the different partial views of the memory different threads can have, and in particular to express they are separated, we base the type-and-effect system on regions. A reference over a value of type int therefore no longer simply state that it points to a cell of type int the memory, but is more precisely indexed by a region: hence if x and y are respectively of type $ref_\rho int$ and $ref_\sigma int$, we know they are references in different regions.

Additionally, regions are split between private and public regions, intuitively stating wether other thread can access them or not. Our typing contexts therefore no longer are restricted to a simple variable context Γ but enriched with two new components: a set Π of public regions and a set Λ of private regions. A typing judgment takes now the following form:

$$\Pi \mid \Lambda \mid \Gamma \vdash\ e :\ \tau,\ \epsilon$$

Having new assumption about the memory environment, we also collect new informations about its evolution: ϵ is a set of effects describing a safe approximation of reads, writes or allocations over regions that e can perform.

Naturally our logical relation have to reflect this enhancement. Worlds will therefore now be finite sets of regions, this subdivision of worlds being usually referred in the literature as a division in islands [1]. Each region is associated in the world to sets of pairs of locations and semantic types, similarly to our previous setup.

Concurrency raises another subtlety: in our previous setup, as it is usual for logical relations, we roughly relate two expressions given they reduce to related values; we refer to this intuition as the *extensional* view. Indeed, we generally do not care about intermediate states. However, concurrency rises a need to care, since computation can be interrupted at any moment, potentially modifying our public regions. Specifically for public regions, the *granularity of extensionality* must therefore be reduced to a single step, while it still can be the entire computation for private regions.

Note that while this is not addressed in the logical relation designed for proving the Parallelization Theorem [6], attempts to work over even richer worlds have been performed. Intuitively, worlds do not describe directly the current state of the heap but rather the invariants on the heap which are expected. With worlds as simple as the ones described in Section 5, these invariants simply state that two locations store values of related type. Richer invariants, specified as protocols, or abstract state transitions can permit one to deal with trickier examples, as Dreyer *et al.* illustrated by introducing a notion of population over the islands composing the worlds [1, 8]. Intuitively, allocation allowed to add an island to the world, but the islands themselves were static objects, which revealed to be inadequate for dealing with a notion such as the one of abstract data type. By adding a notion of population, they manage to keep track of some knowledge about the history of the local state.

7 Conclusion

Relational reasoning about program equivalence for realistic languages is a challenge, especially when combining concurrency with higher order store and dynamic memory allocation. One relevant tool is the one of logical relation, a semantic interpretation of types compatible with the typing judgments. The development of a relevant such relation has been the source of many works during the last decades, progressively managing to cope with increasingly rich features. The difficulty in the definition of worlds in presence of recursive types as been solved. A first Kripke-style model of worlds has been introduced to model expected properties of the heap, allowing one to work with references. This elementary model of worlds has been refined through notions of growing populations, allowing for asserting complex invariants under the form of protocols. The addition of regions allow one to reason on concurrent language.

This evolution led to promising results, as illustrated in particular through the Parallelization Theorem proven by Birkedal *et al.* [6]: they designed a logical relation allowing them to show the correctness of program transformations based on a type-and-effect system, and in particular the semantical equivalence between sequential and parallel execution of expression whose effects do not interfere.

However, this promising result build upon numerous complex layers, which rise an interest for a formalization. Indeed, hidden flaws are hard to categorically dismiss. Moreover, numerous models have been developed following this tradition, making it delicate to spot precisely where are the design choices. A formalization could therefore help to develop useful insights, both for teaching as well as future research purposes. It is one such formalization in Coq which constitutes the heart of my internship.

This formalization is, similarly to the outline of this report, thought to be built up progressively, roughly following this working plan:

1. Formalize the operational semantics for $F_{\mu,ref}$
2. Prove progress and preservation for $F_{\mu,ref}$
3. Develop a semantic model of types for $F_{\mu,reft}$ using the CBUlt formalization and prove the logical relation theorem
4. Generalize to type-and-effect system and prove the logical relation theorem
5. Prove the Parallelization Theorem

At this date, the two first items are completed, and the third is (hopefully[2]) about to be. January shall see a proper breakthrough toward concurrency!

References

[1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. *SIGPLAN Not.*, 44(1):340–353, January 2009.

[2] Amal Jamil Ahmed. Semantics of types for mutable state. Technical report, 2004.

[3] Roberto M. Amadio. Recursion over realizability structures. *Information and Computation*, 91(1):55 – 85, 1991.

[2]If I have already learnt one thing through this internship, it certainly is that the only legitimate "almost finished" Coq proof is a finished Coq proof.

[4] Andrew W. Appel and David Mcallester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23:2001, 2000.

[5] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *Proceedings of POPL 2011*, 2011.

[6] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *Proceedings of CSL*, September 2012.

[7] L. Birkedal, K. Støvring, and J. Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. *Mathematical Structures in Computer Science*, May 2010.

[8] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. *SIGPLAN Not.*, 45(1):185–198, January 2010.

[9] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[10] Jacob Thamsborg and Lars Birkedal. A kripke logical relation for effect-based program transformations. *SIGPLAN Not.*, 46(9):445–456, September 2011.