Predoc Internship

Report

Toward a formalization in Coq of a parallelization theorem for an higher order concurrent language with general references

Author:
Yannick Zakowski

Supervisor:
Lars Birkedal
Filip Sieczkowski
Logics and Semantics group

Abstract

Birkedal *et al.* have recently designed a type-and-effect system allowing for a Parallelization Theorem to hold: a syntactic non-interference of effects inferred implies contextual equivalence of sequential and parallel executions of two expressions.

However, such a result requires to be able to prove contextual equivalence over a language with higher-order functions as well as dynamic memory allocation. Performing so straightforwardly does not seem to be realistic, hence the need for a tool. The one of use here is a logical relation, based on a semantic interpretation of types.

However, the complexity of these models calls for a formalization. Making use of a recent library for solving recursive domain equations, we formalize models, aiming at formalizing the Parallelization Theorem.

1 Introduction

Relational reasoning about program equivalence allows one to reason about the correctness of program transformations, such as compiler optimizations for instance. However, program equivalence is described through the notion of contextual equivalence, which always is excessively challenging to prove due to the quantification over any possible context. Through recent years, many improvements to various methods have been developed in a hope of lifting these techniques to realistic languages: bisimulations, traces, game semantics and Kripke Logical Relations.

In particular, in the domain of Logical Relations, significant progress has been realized during the last decade : the technique of step-indexing allowed for tackling recursive types [2], and has been refined and abstracted through the notion of guarded recursion [6, 5]; ad hoc Kripke models have been built upon increasingly rich worlds [1, 7], and recently a method to solve the proper recursive equation, by working over an appropriate mathematical universe, has been developed [3].

Consequently, these techniques are nowadays able to tackle realistic languages. We present through this report the first logical relation for reasoning about equivalence of a concurrent higher-order ML-like language with higher-order store and dynamic memory allocation [4], introduced by Lars Birkedal, Filip Sieczkowski and Jacob Thamsborg. The relation is used to prove a parallelization theorem: introducing a type-and-effect system, they proved that if two expressions are inferred to have disjoint effects, their sequential execution is contextually equivalent to their parallel execution.

My internship has been dedicated to formalizing in the Proof Assistant Coq this logical relation, its correctness, and using it to prove the parallelization theorem.

The remaining of this report is organized as follows. In Section 2, we present the language under consideration, the type-and-effect system and state the Parallelization Theorem. Section 3 introduces the notion of logical relation and build toward the one we needed by enriching progressively a very simple logical relation. Section 4 briefly describes the ongoing formalization, before concluding in Section 5.

2 A type-and-effect system allowing for a parallelization theorem

2.1 The language

We consider a standard lambda calculus extended with general references and parallel composition whose syntax is given in Figure 1. We will refer to this language as $\lambda_{ref,\ concurrent}$. We assume additionally countably infinite sets of region variables $\mathcal{RV}$ (ranged over by ρ), locations $\mathcal{L}$ (ranged over by

$$
\begin{array}{lll}
\pi & ::= & rd_\rho \mid wr_\rho \mid al_\rho \\
\varepsilon & ::= & \pi_1, \ldots, \pi_n \\
\tau & ::= & 1 \mid int \mid \tau_1 \times \tau_2 \mid ref_\rho \tau \mid \tau_1 \rightarrow_\varepsilon^{\Pi,\Lambda} \tau_2 \\
v & ::= & x \mid () \mid < v_1, v_2 > \mid fun\ f(x).e \mid l \\
e & ::= & v \mid proj_i\ v \mid v\ e \mid ref\ v \mid !\ v \mid v_1 := v_2 \mid par\ e_1\ and\ e_2 \mid cas\ (v_1, v_2, v_3) \\
E & ::= & [] \mid v\ E \mid par\ E\ and\ e_2 \mid par\ e_1\ and\ E
\end{array}
$$

Figure 1: Syntax

$$
\begin{array}{rcl}
(E[proj_i\ (v_1, v_2)] \mid h) & \longmapsto & (E[v_i] \mid h) \\
(E[(fun\ f(x).e)\ v] \mid h) & \longmapsto & (E[e[fun\ f(x).e/f][v/x]] \mid h) \\
(E[ref\ v] \mid h) & \longmapsto & (E[l] \mid h[l \mapsto v]) \text{ if } l \notin dom(h) \\
(E[v_1 := v_2] \mid h) & \longmapsto & (E[()] \mid h[l \leftarrow v]) \text{ if } l \in dom(h) \\
(E[!l] \mid h) & \longmapsto & (E[h(l) \mid h) \text{ if } l \in dom(h) \\
(E[par\ v_1\ and\ v_2] \mid h) & \longmapsto & (E[< v_1, v_2 >] \mid h) \\
(E[cas\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[1] \mid h[l \leftarrow n_2]) \text{ if } l \in dom(h) \text{ and } h(l) = n_1 \\
(E[cas\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[0] \mid h) \text{ if } l \in dom(h) \text{ and } h(l) \neq n_1
\end{array}
$$

Figure 2: Semantics

l) and program variables (ranged over by x, f). Heaps are modeled as finite maps from locations to values. We note $FRV(e)$ for the set fo free region variables in e.

We provide a very standard small step semantics between configurations, constituted of an expression and a heap, as can be found on Figure 2. Note that for convenience, we keep the amount of expression contexts minimal, but this is of no theoretical restriction: pairs for instance can be encoded through lambda abstraction as $< e_1, e_2 >::= (\lambda\ x,\ y.\ < x, y >)\ e_1\ e_2$. A parallel execution can non-deterministically reduce either expression, and reduce to a pair of value once both expressions reduced down to a value.

2.2 Type-and-effect system

As one can have already noted by scheming through the syntax of types, we do not use a simple type system but a so-called type-and-effect system. Indeed, the intuitive idea is that we not only want to track the type of an expression, but also an over approximation of the effects that this expression might have during its execution: reads, writes and allocations. Compared to usual type-and-effect systems, regions are partitioned into public and private regions, leading to a judgment of the following form:

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon.$$

As usual, Γ is a variable context, mapping types to program variables, e an expression and τ the resulting type assigned to e. Regions describe an

abstract view of the heap, which is here split between public ones in Π and private ones in Λ. Intuitively, a private region is a piece of memory over which we have exclusive ownership, while public regions are shared by several threads. Alternatively, one can think about it in term of rely/guaranty vocabulary: you rely on the fact that the environment will only read, write and allocate, in a well type manner, over public regions, but won't touch your private regions, and you guarantee that you will write exclusively over regions in your private list.

Finally, we track accesses to the memory by inferring a list of effects of the form rd_ρ, wr_ρ and al_ρ where ρ is a region to infer an over approximation of the set of read, writes and allocations the execution of e might entail.

The complete type-and-effects system is displayed on Figure 3. The rule for parallel threads illustrates the use of private and public regions: regions which was private on top level become public when typing both subexpressions: indeed, each of them have access to these regions, they hence are public from this perspective. The masking rule allows for introducing new private regions:

$$\frac{\Pi \mid \Lambda, \rho \mid \Gamma \vdash e : \tau, \varepsilon \qquad \rho \notin FRV(\Gamma, \tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon - \rho}$$

This rule can be thought in a similar fashion as frame rules are in separation logics: it allows for locally introducing a private regions, and remove any mention of it, including the effects over it, once back in the general context. The rule is ensure to be sound through the side condition: e has no way from leaking locations from ρ.

Note how the rules for assignments, allocations and lookups make use of the region annotation on reference types, marking where the location lives. And finally, the arrow type, despise looking a bit intricate, is very natural: a lambda abstraction being essentially a suspended computation, we require a similar setup to the general type judgment: we annotate the expected public and private regions by the function, and the set of effects it might have.

2.3 The Parallelization Theorem

We define (may-)equivalence the usual way: intuitively, two expressions are said to be contextually equivalent if any context they are put in can't observe their difference.

Definition 1. *Contextual equivalence.*

$\Pi \mid \Lambda \mid \Gamma \vdash e \leq_C e' : \tau, \varepsilon$ *if and only if for all heap* h *and well typed context* C*, whenever* $(C[e]|h)$ *terminates then* $(C[e']|h)$ *terminates.*

Contextual equivalence, $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \approx_C e_2 : \tau, \varepsilon$*, is defined as contextual approximation in both directions.*

$$\frac{}{\Pi \mid \Lambda \mid \Gamma, x:\tau \vdash x:\tau, \emptyset} \qquad \frac{}{\Pi \mid \Lambda \mid \Gamma \vdash () : 1, \emptyset} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash v : \tau_1 \times \tau_2, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash proj_i : \tau_i, \varepsilon}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash v_1 : \tau_1, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash v_2 : \tau_2, \varepsilon_1}{\Pi \mid \Lambda \mid \Gamma \vdash < v_1, v_2 > : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Pi \mid \Lambda \mid \Gamma, f : \tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2, x : \tau_1 \vdash e : \tau_2, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash fun\ f(x).e : \tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2, \emptyset}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash v : \tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e : \tau_1, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash v\ e : \tau_2, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash v : \tau, \varepsilon \qquad \rho \in \Pi, \Lambda}{\Pi \mid \Lambda \mid \Gamma \vdash ref\ v : ref_\rho \tau, \varepsilon \cup \{al_\rho\}} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash v_1 : ref_\rho \tau, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash v_2 : \tau, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash v_1 := v_2 : 1, \varepsilon_1 \cup \varepsilon_2 \cup \{wr_\rho\}}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash v : ref_\rho \tau, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash !v : \tau, \varepsilon \cup \{rd_\rho\}} \qquad \frac{\Pi, \Lambda \mid . \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda \mid . \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash par\ e_1\ and\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash v_1 : ref_\rho int, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash v_2 : int, \varepsilon_2 \qquad \Pi \mid \Lambda \mid \Gamma \vdash v_3 : int, \varepsilon_3}{\Pi \mid \Lambda \mid \Gamma \vdash cas\ v_1\ v_2\ v_3 : int, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \{wr_\rho, rd_\rho\}}$$

$$\frac{\Pi \mid \Lambda, \rho \mid \Gamma \vdash e : \tau, \varepsilon \qquad \rho \notin FRV(\Gamma, \tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon - \rho}$$

Figure 3: Type-and-effect system

Note that contextual equivalence actually enforce not only equivalence of termination but also that both expressions reduce to the same value: indeed, by quantifying over all context, we always can wrap them around a loop which would terminate only over a specific value.

The type-and-effect system allows for a parallelization theorem: intuitively, if two well typed expressions have disjoint effects, then their parallel and sequential executions are contextually equivalent. Formally, we get:

Theorem 1. *Parallelization Theorem.*
Assuming that

1. $\Pi, \Lambda \mid . \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$,
2. $\Pi, \Lambda \mid . \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$,
3. $rds\ \varepsilon_1 \cup wrs\ \varepsilon_1 \cup rds\ \varepsilon_2 \cup wrs\ \varepsilon_2 \subseteq \Lambda$,
4. $rds\ \varepsilon_1 \cap wrs\ \varepsilon_2 = rds\ \varepsilon_2 \cap (wrs\ \varepsilon_1 \cup als\ \varepsilon_1) = wrs\ \varepsilon_1 \cap wrs\ \varepsilon_2 = \emptyset$,

we get the following property:

$$\Pi \mid \Lambda \mid \Gamma \vdash < e_1, e_2 > \approx_C par\ e_1\ and\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$$

The hypotheses are essentially what we would expect: item 3 prevents the environment from observing anything by setting every region written or

read as private on the top level, and item 4 prevents both computations to interact, rendering potential interleavings in the parallel case observationally equivalent. The asymmetry introduced by the $als\ \varepsilon_1$ is an unsatisfactory but necessary technicality.

Note that this optimization is based on an entirely syntactic judgment, a type-system, it hence could be performed automatically and systematically by a compiler. Additionally, the masking rule allows for many more uses of the Parallelization Theorem by allowing to abstractly see effectful computations as pure.

However, proving such a theorem is challenging. Indeed, it states a contextual equivalence, we hence need to prove something quantified over any possible context. Additionally, it takes place in a concurrent setting, requiring us to consider any possible interleaving. A straightforward attempt at such a proof would hence be bound to fail: as stated in the introduction, we need to rely on a more involved technique, and choose here the one of logical relation.

3 A logical relation for $\lambda_{ref,\ concurrent}$

If a straightforward structural induction might sometimes is enough, to prove type safety through the usual progress and preservation lemmas for instance, semantically interesting properties often require us to find a stronger induction principal. The case of strong normalization for simply typed lambda-calculus illustrates quite efficiently this idea.

Theorem 2. *Strong normalization If* $\Gamma \vdash t : \tau$ *then* t *reduces to a value under a finite among of steps.*

If one tries to prove this results by structural induction, the case of the β-reduction is quite problematic. Indeed, we need to prove that $t_1\ t_2$ is normalizable. By induction and inversion of the typing judgments, $t_1 \rightarrow^* \lambda\ x.t$ and $t_2 \rightarrow^*\ v_2$. However, we are left with arguing that $t[v_2/x]$ is normalizable. But this term might be bigger than the initial one, since x might appear several times in t, we hence are stuck.

One possible tool to solve this problem, introduced by Girard in order to prove strong normalization of System F, is the one of Logical Relation. Schematically, the idea follows these three steps:

- we provide a semantic interpretation of types over a semantic universe, typically subsets of syntactic values;
- we lift this interpretation over expressions;
- finally, we define a logical relation over it, similar to a typing judgment, essentially as membership to the corresponding interpretation of type.

This relation can either be unary, essentially relating expressions at types they have, or binary, essentially relating together contextually equivalent expressions. In the second case, we define an auxiliary unary judgment as being related to itself. While many such logical relations has been defined over the last decades, allowing for enforcing various invariants, two main theorems are always required for them to be granted as sound:

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

The non-blocking property being encompassed in the interpretation of types, we recover type safety "for free" from this theorem. Its proof goes by induction over the type judgment, using so-called compatibility lemmas: for each typing rules, we prove it to be a theorem if read with the logical relation instead of the typing judgment.

Theorem 4. *Soundness.*
If $\Gamma \vDash (t_1, t_2) : \tau$ *then* $\Gamma \vdash t_1 \approx_C t_2 : \tau$.

And what is generally of core interest for us: logical relation entails contextual equivalence. The relation generally won't be complete, but will allow for proving more easily contextual equivalence of classes of expressions. This property is the one we need for our Parallelization Theorem: under the hypotheses of theorem 1, we will prove that e_1 and e_2 are logically related rather than directly contextually equivalent.

In the remaining of this section, we built toward the desired logical relation by restricting ourself to the simply typed λ-calculus as a first step, and then introduce successively references and concurrency in our language.

3.1 A logical relation for the simply-typed λ-calculus

No much thought need to be given to choose the universe of semantic types: for such a language, it is possible to simply interpret types as set of syntactic values. As detailed in figure 4, the interpretation of base types is defined as syntactic equality. However the arrow case is more interesting: we relate two λ-abstractions if and only if, applied to related arguments, they lead to related results. However, after substitution, we end up with expressions and no more values, we hence need to formalize the idea of "leading to" related values. This is achieve by lifting the interpretation to expressions: two expressions are related if every time the first one normalizes to a value, the other one also normalizes to a value and both are related.

Note that if our purpose was the proof of strong normalization, we would have built-in the definition of the lifted interpretation the requirement for expressions to be strongly normalizable.

$$
\begin{array}{rl}
[\![1]\!] & = \{((),())\} \\
[\![int]\!] & = \{(n,n) \mid \forall n \in \mathbb{N}\} \\
[\![\tau_1 \times \tau_2]\!] & = \{(< v_{1,1}, v_{1,2} >, < v_{2,1}, v_{2,2} >) \mid (v_{1,1}, v_{2,1}) \in [\![\tau_1]\!] \wedge (v_{1,2}, v_{2,2}) \in [\![\tau_2]\!]\} \\
[\![\tau_1 \to \tau_2]\!] & = \{(\lambda x.e_1, \lambda x.e_2) \mid \forall (v_1, v_2) \in [\![\tau_1]\!],\ ((\lambda x.e_1)\ v_1, (\lambda x.e_2)\ v_2) \in \mathcal{E}[\![\tau_2]\!]\} \\
\mathcal{E}[\![\tau]\!] & = \{(e_1, e_2) \mid e_1 \to^* v_1 \Rightarrow \exists v_2,\ e_2 \to^* v_2 \wedge (v_1, v_2) \in [\![\tau]\!]\}
\end{array}
$$

Figure 4: Interpretation of types for the simply typed λ-calculus

3.2 A logical relation for λ_{ref}

The introduction of references in our language rises a new question: how shall we interpret the type of references over a type τ? Intuitively, thinking first of a unary model, we would like it to be the set of locations containing values in the interpretation of τ, hence we would like to write something among those lines:

$$[\![ref\ \tau]\!] = \{l \mid h(l) \in [\![\tau]\!]\}$$

However, this highlights a new necessity: the interpretation of types now depends on the current state of the heap. We hence need to parametrize the interpretation of types by worlds representing an abstract view of the heap, asserting an invariant the heap must satisfy. Our new semantic universe hence looks like so:

$$T = W \to^{mon} Pred(Val)$$

Note that we ask this mapping to be monotone: this is a way to ensure that invariants we enforce are maintained through a growing heap.

However, the question now is to define the worlds. They are meant to model the heap by specifying an invariant any heap modeled by a world has to satisfy. The simplest we can think of hence is to state that locations contains values of a fixed type. This leads to the following universe of worlds:

$$W = Loc \to T$$

But these definitions are recursive! If we unfold one in the other, we get the following recursive domain equation:

$$T \cong (Loc \to T) \to^{mon} Pred(Val)$$

This equation does not admit any solution in the category Sets as a cardinality argument highlights. We therefore need more structure, to restrict ourself to a more specific category. There are several equivalent ways to achieve this goal, but one is to work over the category of Complete 1-Bounded Ultrametric space, equipped with non-expensive maps. We denote this category as $CBUlt_{ne}$.

Definition 2. *An ultra metric space is a metric space whom distance satisfies the following stronger triangular inequality:*

$$d(x,z) \leq max(d(x,y), d(y,z))$$

Such a space is said to be complete if every Cauchy sequence has a limit, and 1-bounded if its distance function takes value in $[0, 1]$.

A map $f : (U, d_1) \to (V, d_2)$ *is said to be non-expensive if* $\forall x, y \in U, d_2(f(x), f(y)) \leq d_1(x, y)$. *It is said to be contractive if* $\delta < 1$ *such that* $\forall x, y \in U, d_2(f(x), f(y)) \leq \delta.d_1(x, y)$

Over this category, one can prove a Banach-like fix-point theorem: any (locally) contractive function $F : CBUlt_{ne}^{op} \times CBUlt_{ne} \to CBUlt_{ne}$ admits a unique (up to isomorphism) fix-point.

In order to equip our setup with such a mathematical structure, we reuse a facility introduce in the interpretation of recursive types[1]: step-indexing. The intuitive idea is to no longer relate contextually equivalent expressions, but expressions whose behavior is identical for at least a certain amount of computational steps. This allows us to introduce a metric between our set of predicates, two sets being intuitively closer the more steps we can allow ourself while still letting them coincide.

Formally, we do not work over predicates of values but so-called *uniform* predicates of values: sets of pairs of natural numbers and values which are downward closed in the first component - if they agree for n steps, they also agree for less than n steps:

$$UPred(Val \times Val) = \{p \subseteq \mathbb{N} \times Val \times Val \mid \forall (k, v_1, v_2) \in p, \forall j \leq k,\ (j, v_1, v_2) \in p\}.$$

Then, by defining the approximation of a uniform predicate as $p_{\lfloor k \rfloor} = \{(i, v_1, v_2) \in p \mid i < k\}$, we equip this space with a bisected metric:

$$d(p, q) = \begin{cases} 2^{-max\{k \mid p_{\lfloor k \rfloor} = q_{\lfloor k \rfloor}\}} & \text{if } p \neq q \\ 0 & \text{otherwise} \end{cases}$$

Equipped with this distance, $(UPred(Val \times Val), d)$ can be proved to be an object in $CBUlt_{ne}$. We hence can solve our recursive equation over this space. A last adjustment must be performed since we have motivated our worlds while having a unary setup in mind. We now do not want to associates types to locations, but to relate locations at types. We hence defined our worlds as finite partial bijections from locations to locations annotated with types. Formally, it hence is a triple of finite maps, two of them going from locations to locations and being inverse of each other, and one going from locations to semantic types. We hence actually solve the following equation:

$$\hat{T} \cong 1/2(ParBij(\hat{T})) \to^{mon} UPred(Val \times Val)^2$$

[1] We do not cover recursive types here in order to lighten a bit an already quite heavy formalism. However, they are an orthogonal feature which could be added to the model. Similarly, universal and existential types could be added and, contrary to the recursive types, do not rise much difficulty.

[2] The one half is a very common mathematical trick to get a contractive map out of a non-expensive one.

Using Banach's fix point theorem, we hence use our solution to define our universe of worlds as $W = ParBij(\hat{T})$ and our semantic universe as $T = W \to^{mon} \hat{T}$. We also obtain an isomorphism ι/ι' between T and $\hat{T}$ allowing us to safely tie the loop.

While this can appear to be quite a burden, the method actually has great advantages. First, the method is extremely general: indeed, this kind of recursive domain equations appear constantly when trying to model realistic features, and the technic is by no mean tied to this specific model. Additionally, the resolution of the equation, where all the technicalities are hidden, can be used as a black box. The user simply has to provide the adequate mathematical structure[3] to the domain he is working over. Once this step is passed, the modeled can be defined without worrying much about the origin of these universes. And finally, the additional structure actually acts as a guide in building the model: indeed, building ad hoc models for such languages is quite an art, decrementing the adequate index in the adequate place requiring much insight and experience. Having to explicitly enforce non-expensiveness at every step of our definition actually carry out some of these insights and act as a safe-net.

We are now ready to interpret our types, as displayed in figure 5. Notice the use of the approximation of uniform predicates in the interpretation of reference types, allowing it to be well-defined: we only require locations to be related at a type behaving similarly to the interpretation of τ for k steps. The arrow type is very similar to its previous interpretation, except that we explicitly downward-close it, and that we quantify over any possible futur world: a intuitive explanation of this fact comes down once again to lambda abstractions being suspended computation, we hence want to preserve the invariant when the computation actually takes place, which might be after new allocations. Finally, the evaluation closure makes explicit use of the step indexing: if we take i steps of computations, we lower the requirement of relatedness by the same amount. We also need to introduce a new feature: concretization of worlds. Indeed, we need to establish the property for a computation starting from any heap concretizing w, and reestablish the existence of a futur world abstracting the resulting heap. This can be read as *rely/guarantee* statement: we rely on the environment being modeled by w, and we guarantee that at the end of our computation, the memory can be modeled back by a world extending the initial one.

Note that behind the scenes have to take place quite a few proofs: since we work over a specific universe, we have to make sure that what we write down actually lives in this universe. We hence notably need to prove monotony and non-expansiveness of each of these interpretations of types. It is possible to ease this burden by working directly in an adequate

[3]Alternatively, one can work over complete ordered family of equivalence instead of $CBUlt_{ne}$.

$$
\begin{array}{rll}
[\![1]\!] & w & = \{(k, (), ())\} \\
[\![int]\!] & w & = \{(k, n, n) \mid \forall n \in \mathbb{N}\} \\
[\![\tau_1 \times \tau_2]\!] & w & = \{k, (< v_{1,1}, v_{1,2} >, < v_{2,1}, v_{2,2} >) \mid \\
 & & \qquad (k, v_{1,1}, v_{2,1}) \in [\![\tau_1]\!]\ w \wedge (k, v_{1,2}, v_{2,2}) \in [\![\tau_2]\!]\ w\} \\
[\![ref\ \tau]\!] & w & = \{(k, l_1, l_2) \mid (l_1, l_2, \mu) \in w \wedge \forall w' \sqsupseteq w, (\iota\mu\ w')_{\lfloor k \rfloor} = ([\![\tau]\!]\ w')_{\lfloor k \rfloor}\} \\
[\![\tau_1 \to \tau_2]\!] & w & = \{(k, \lambda x.e_1, \lambda x.e_2)\ | \forall i \leq k, \forall w' \sqsupseteq w, \forall (i, v_1, v_2) \in [\![\tau_1]\!] w'\} \\
 & & \qquad (i, (\lambda x.e_1)\ v_1, (\lambda x.e_2)\ v_2) \in \mathcal{E}[\![\tau_2]\!] w'\} \\
\mathcal{E}[\![\tau]\!] & w & = \{(k, e_1, e_2) \mid \forall i \leq k, \forall h :_k w,\ (e_1 \mid h) \to^i (v_1 \mid h') \\
 & & \qquad \Rightarrow \exists v_2,\ e_2 \to^* v_2 \wedge \exists w', h :_{k-i} w' \wedge (k - i, v_1, v_2) \in [\![\tau]\!]\}
\end{array}
$$

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

Figure 5: Interpretation of types for λ_{ref}

model such as the topos of trees, where intuitively everything one can write will be *de facto* non-expansive, but requires from the user more categorical background [5].

3.3 A logical relation for $\lambda_{ref,\ concurrent}$

Building up, the issue is now to handle concurrency. The semantic universe stay essentially similar: we simply need to introduce the notion of region through our worlds. A world hence is now a finite map from region names to options over partial bijections annotated with semantic types:

$$W = \mathcal{RN} \rightharpoonup_{fin} Option(ParBij\ T).$$

We additionally require the regions to be disjoint: two different regions associated to actual bijections can't both talk about a same location. The *Option* allows region names to be in three potential states: undefined, alive or dead. Worlds can hence evolve in three distinct ways: a fresh region name can be set alive and associated with an empty partial bijection; a live region can be killed, losing its associated partial bijection; or a live region can be extended with a new triple, granted the disjointness condition is not broken.

We hence reuse the same method as previously to define our world and semantic types universes as solution of a recursive domain equation. There are a bit more work to do in order to equip these worlds with the desired structure of $CBUlt_{ne}$, but no additional theoretical difficulty.

However, when it comes to interpreting types, things get a bit more involved. Indeed, Figure 6 describes the previous situation. We have an abstract vision of the memory, an invariant, and a way to concretize it down to a heap. Starting from any adequate heap, we need to exhibit an extending world modeling the final heap. However, now that we consider a concurrent

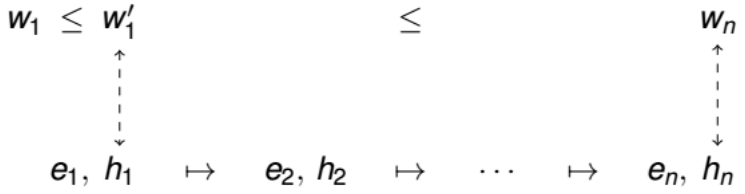

Figure 6: Requirements in a sequential setting.

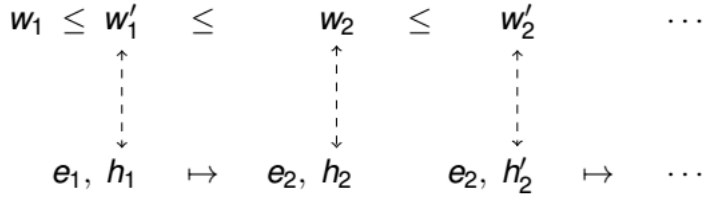

Figure 7: Requirements in concurrent setting.

setup, the environment can shake up our situation at any time: we need a finer-grained model!

As exhibited through Figure 7, we need to relate our heap to a world at every step: it is our guarantee to the environment, we state that there exist a futur world matching our heap. But it then is the environment's turn to interfere: the next step of computation must start from a heap matching any world the environment could have transited to, we only rely on this world to extend the previous one. Note that in general, these two notions of extension of worlds have no reason to be the same, since one describes what we can do, while the other describes what the environment can perform.

Formally, this distinction intervenes through the evaluation closure, which gets significantly more complicated. We replace the interpretation over expression by a *safe* predicates, reproduced in Figure 8. We provide it essentially as an illustration of the complexity[4] of the relation and do not provide most of notations used, but will rather describe it intuitively.

Safety between two expression e_1 and e_2 essentially states that, after interference of the environment, the behavior of e_1 can be matched by e_2, and adequate relatons can be reestablished afterwards: safety itself, existence of a futur world modeling the new concrete memory notably... The predicate is therefore divided in three main parts, surrounded by brackets. The first one expresses the interference of the environment, yielding to a new world. The second one is the termination branch, it describes the expectations we have on e_2 if the behavior of e_1 happens to be termination. The last one is the progress branch, the case where e_1 takes one step, and we can notably note

[4]The symbols in bold are auxiliary predicates not defined here.

that we calls recursively the safe predicates, decrementing the step index in the process. Note that the predicate **envtran** unfolds to the description of the transition relation in Figure 7 used by the environment, while **selftran** unfolds to ours.

Nonetheless, the interpretation of types itself is very similar in substance to the one for λ_{ref} once equipped with the *safe* predicate instead of the $\mathcal{E}[\![\tau]\!]$ predicate. The relation can then be proved to be sound, and exploited to prove the Parallelization Theorem.

4 Formalization in Coq

My internship, set aside pedagogical purposes, aimed at formalizing in Coq the Parallelization Theorem. This objective is rendered possible thanks to a library, developed mainly by Filip Sieczkowski, formalizing the notion of $CBUlt_{ne}$ and the resolution of domain recursive equations over this category. The library makes intensive use of the quite recent Coq feature of *type classes*. It essentially is a record (which, in Coq, can contains proofs in place of fields), equipped with proof search facilities. This allows for building nicely incrementally rich structures such as here for instance, the notions of spaces equipped with equivalence relations, metrics, complete metric spaces, partially ordered complete metric spaces... Having for instance provided a process to equip the cartesian product of two metric spaces with a structure of metric space itself, the proof search facility will be able to infer by himself such a proof any time the structure would happen to be required.

If very convenient, this feature does not come with explicit error messages. It therefore require quite a decent understanding of their use, and more importantly, an excellente comprehension of what we are currently trying to formalize. I therefore experimented this library through the formalization and proof of soundness of a model for λ_{ref}, similar to the one presented in Section 3.2. Aside from a necessary step in learning the required material, both on the theoretical side and the Coq side, it contributed to enrich the library and helped in the design of an ongoing tutorial Filip is creating[5].

I then step on the proper objective: $\lambda_{ref,\ concurrent}$. However, the complexity significantly blows up. Modeling partial bijections and worlds, equipping them with the adequate structure and solving the recursive domain equation required a refactoring of the library to improve the instances system, Coq tending to get stuck in endless type checking or to simply crash. We however eventually managed to get it right, and it now satisfactorily takes approximatively 45 minutes to compile on my laptop.

The next major difficulty has been the definition of the safety predicate itself. It requires numerous proofs of non-expansiveness, monotonicity

[5] `http://cs.au.dk/~birke/modures/tutorial/index.html`

$$
\begin{array}{l}
(k, h_1^\circ, h_2^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\, w^\circ, w \\
\quad \iff \\
\forall j \le k.\, \forall w', g_1, g_2, f_1, f_2. \\
\quad \Big[\mathbf{envtran}^{\Pi,\Lambda,A,R}\, w, w' \wedge (j, g_1, g_2) \in \mathbf{P}_\varepsilon^{\Pi,R}\, w' \wedge \\
\quad (g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R}\, w'\Big] \Rightarrow \\
\quad \Big[\mathrm{irr}(e_1 | g_1 \cdot h_1 \cdot f_1) \Rightarrow \\
\quad\quad \exists e_2', w'', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \mid g_2 \cdot h_2 \cdot f_2) \longmapsto^* (e_2' \mid g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad \emptyset = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge g_1 \cdot h_1 = g_1' \cdot h_1' \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\emptyset,R}\, w'' \wedge (j, g_1, g_2, g_1', g_2') \in \mathbf{Q}_\varepsilon^{\Pi,R}\, w', w'' \wedge \\
\quad\quad\quad (j, e_1, e_2') \in [\![\tau]\!]^R\, (w'') \wedge \exists h_1'' \subseteq h_1', h_2'' \subseteq h_2'.\, (j, h_1^\circ, h_2^\circ, h_1'', h_2'') \in \mathbf{Q}_\varepsilon^{\Lambda,R}\, w^\circ, w''\Big] \wedge \\
\quad \Big[\forall e_1', h_1^\dagger, \mu, n \le j.\, (e_1 \mid g_1 \cdot h_1 \cdot f_1) \to_\mu^n (e_1' \mid h_1^\dagger) \Rightarrow \\
\quad\quad \exists e_2', w'', A', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \mid g_2 \cdot h_2 \cdot f_2) \longmapsto^* (e_2' \mid g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad A' = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge h_1^\dagger = g_1' \cdot h_1' \cdot f_1 \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,A',R}\, w'' \wedge (j - n, g_1, g_2, g_1', g_2') \in \mathbf{Q}_\varepsilon^{\Pi,R}\, w', w'' \wedge \\
\quad\quad\quad \mu \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R}\, w'' \wedge (j - n, h_1^\circ, h_2^\circ, e_1', e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A',R}\, w^\circ, w''\Big]
\end{array}
$$

Figure 8: Safety

and compatibility with equivalence. However, passed this consequent step, the remaining definition of the model, the interpretation of types itself, is relatively easy.

The formalization of the syntax, semantics and type system is essentially straight forward.

However, I still have quite a long road to follow. I am at the moment at this step, and need now to prove the correctness of the model. While conceptually simple, the proof shall be quite tedious and require a bit of work.

Once this is done, the Parallelization Theorem should finally be proved. However, this proof will not only require Coq hacking, but also some theoretical insight. Intuitively, the difficulty is that in the parallel execution, a reduction step taken by the second expression cannot be matched immediately by itself in the sequential execution, since the first expression must reduce down to a value before. We hence need to introduce a notion of potential-safety, suspending this match after other transitions to be determined.

5 Conclusion

During the last decades, numerous efforts have been put to develop technics to reason about program equivalence in the context of realistic languages. Among these techniques, the one of Logical Relations have seen great improvements: step-index models allowed for handling recursive types, Kripke models allowed for tackling references, and recent work over recursive domains equations allowed to tackle faithfully and abstractly this feature.

Coupled with a growing general understanding of the couple *sharing/ownership*, concurrency is nowadays actually manageable. Birkedal *et. al* illustrate this status by providing the first proof of a Parallelization Theorem based on a type-and-effect system for a concurrent higher order language with general references.

However, the complexity of such models, sometimes resting on a decent amount of mathematics, rise a need for their formalization. Through my internship, I got to discover this domain, formalize in Coq and prove sound a model for λ_{ref} and formalize the model introduced in [4]. The work is however not finished, soundness remaining to be proved, and the model to be actually used to prove the Parallelization Theorem.

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] 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.

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

[5] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012.

[6] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

[7] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. *SIGPLAN Not.*, 45(1):171–184, January 2010.