Active Objects based on Algebraic Effects

Martin Andrieux, Ludovic Henrio, Gabriel Radanne

Active Objects based on Algebraic Effects

Martin Andrieux[1], Ludovic Henrio[2], and Gabriel Radanne[2]

[1] ENS Rennes, France
martin.andrieux@ens-rennes.fr
[2] Université Lyon, EnsL, UCBL, CNRS, Inria, LIP, France
{ludovic.henrio,gabriel.radanne}@ens-lyon.fr

Abstract. Algebraic effects are a long-studied programming language construct allowing the implementation of complex control flow in a structured way. With OCaml 5, such features are finally available in a mainstream programming language, giving us a great opportunity to experiment with varied concurrency constructs implemented as simple libraries. In this article, we explore how to implement concurrency features such as futures and active objects using algebraic effects, both in theory and in practice. On the practical side, we present a library of active objects implemented in OCaml, with futures, cooperative scheduling of active objects, and thread-level parallelism. On the theoretical side, we formalise the compilation of a future calculus that models our library into an effect calculus similar to the primitives available in OCaml; we then prove the correctness of the compilation scheme.

1 Introduction

A *future* [1, 9] is a standard synchronisation artefact used in programming languages with concurrency. It provides a data-flow oriented synchronisation at a higher level of abstraction than locks or monitors. A future is a promise of a result from a spawned task: it is a cell, initially empty, and filled with a value when the task finishes. Accessing this value synchronises the accessor with the end of the task. Promises [17] is a notion similar to futures except that a promise must be filled explicitly by the programmer. Promises are more flexible but also more difficult to use because one could try to fill a promise several times and this raises many issues.

Future pay a crucial role in the implementation of asynchronous computations, particularly in object-oriented languages. ABCL/f [23] proposed the first occurrence of typed futures as a mean for *asynchronous method invocation*, where a spawned task fills the future later. Then Creol [13] and ProActive [4] introduced *active objects* [3]; which are both an object (in the sense of object oriented programming) and an actor. As a consequence, an active object has its own logical thread and communications between active objects is done by asynchronous method invocations, using futures to represent the result of asynchronous calls.

Futures, promises, and concurrency primitives in general, have been implemented using a wide variety of techniques, often via dedicated runtime support.

Many concurrency primitives require suspending, manipulating and resuming arbitrary computations. This need for non-local control flow appears as soon as task scheduling is not trivial. It turns out that *effect handlers* [2] precisely enable users to define new control-flow operators. This was quickly identified as a potential technique to implement concurrency primitives while developing Multicore OCaml. Multicore OCaml [21, 22] is an ensemble of new features, including effect handlers, which enable parallel and concurrent programming in OCaml. Crucially, since effects are user-defined , they allow implementing concurrency operators such as futures as *libraries*. This remark is not a contribution of this article, and seems to be well known folklore among algebraic effects practitioners. It is also how `eio` [15], the new concurrency library for OCaml, is implemented.

This article expands beyond this folklore in two direction: First, we showcase how to use effects to implement other concurrency primitives, active objects, that were not previously explored. Second, we formalise the translation between futures and algebraic effects, and prove it correct.

Contribution 1: An actor library based on algebraic effects On the practical side, we present *a new implementation of active objects based on algebraic effects*. It takes the form of an OCaml library that features all the characteristics of active objects, adapted to the OCaml ecosystem. The implementation heavily relies on effect handlers. The library is presented in Section 3.

Our implementation only requires effect handlers and objects. To our knowledge and at the time of writing, both are only conjointly present in OCaml. However, as effect handlers are gaining interest and are developed in different contexts, we believe our methodology is applicable to develop active object libraries in any language that support both features.

Contribution 2: A formalised translation from actors into effect handlers and its proof of correctness On the theoretical side, we present the formal arguments showing that our implementation of active objects by effect handlers fully follows the paradigms of active object languages, more precisely:

- In Section 4 we describe our calculi: first, an imperative λ-calculus similar to what can be found in the literature; second, FUT, which expands this λ-calculus with operations on futures: *parallelism, tasks, futures, cooperative scheduling inside a thread*; finally, EFF which expands the λ-calculus with *effects and effect handlers* in a *parallel* setting.
- Section 5 defines *a compilation scheme from FUT to EFF* that expresses the principles of the implementation of our active object library based on effects. We formally prove the correctness of our translation and show that the behaviours of the effect compilation of futures mimics exactly the future semantics. Our main theorem states that the compiled program faithfully behaves like the original one.

n

2 Context and Positioning: Futures, Promises, Effects

We start by revisiting, in a streamlined fashion, the context of our works. We first present the formal models that exist to define semantics of futures and effects, explaining why we need a new semantics to formalise our work. Then we present the programming patterns we rely on: an API for promises as it would appear in OCaml, and how to implement such API using algebraic effects.

2.1 Formal Model for Futures and Effects

In order to formalise our translation, we need a calculus modeling the core of active objects. Compared to existing active object languages, we base our work on a simple λ-calculus enhanced with imperative operations and futures featuring cooperative scheduling. This calculus does not reflect the object-oriented nature of active object languages. Indeed, while the object layer provides an effective programming abstraction and strong static guarantees, we are mostly interested in operational aspects where objects play little role. Conversely, we consider that cooperative scheduling is essential, as it precisely captures the dynamic behavior we want to reproduce after translation to algebraic effects.

Among previously existing calculi, we position ourselves compared to the following ones. On one hand, several previous calculi [7, 6] rely on a pure λ-calculus, lacking any imperative features. We consider modeling imperative code essential, as it allows us to encode the stateful nature of active objects. In particular pure calculi are not able to represent cycles of futures [10]. On the other hand, a concurrent λ-calculus with futures [18] and the DeF calculus [5] feature imperative aspects but no cooperative scheduling, which is crucial to many active objects languages. Additionally, DeF separates cleanly global state and local variables and uses a notion of functions closer to object methods instead of λ-calculus. We do not believe these features are needed in our context. Finally, some formalisation efforts such as ABS [14] cover much more ground, including a full-blown object system and "concurrent object groups" to model the concurrent semantics. We believe such semantics can also be modeled by simpler mechanisms, such as threads and remote execution of pieces of code.

In the remaining of this work, we use a minimal λ-calculus that includes the following features, that are, from our point of view, the core runtime characteristics of actors and active object languages with futures:

- Impure λ-calculus with a store and memory locations,
- Cooperative scheduling among tasks on the same parallelisation entity.
- Request-reply interaction mechanism based on asynchronous calls targeting a given thread, and replies by mean of futures. Without loss of generality, asynchronous calls are simply performed by remote execution of a given λ-calculus expression.

One crucial aspect of actors and active objects that we omit in this work is the separation of the memory into separate entities manipulated by a single thread (like e.g. ABS "concurrent object groups"). While this feature is crucial and allows reasoning about the deterministic nature of some active object languages [11]

we would not use it in our developments. We also believe this crucial separation could be added by separating the memory in our configurations into a single memory per thread, either syntactically or using some kind of separation logic.

On the algebraic effect side, we use an imperative λ-calculus with *shallow* effect handlers, similar to Hillerström and Lindley [12]. This fits well with OCaml, which supports both imperative and functional features. Note that both deep and shallow handlers are available in OCaml.

2.2 Promises in OCaml

Promises are not a new addition to OCaml. Historically, the libraries *Lwt* [24] and *Async* implemented monadic promise-based cooperative multitasking in OCaml. Due to OCaml's limitation at the time, neither library implemented parallelism. Multicore OCaml introduced parallelism (with a new garbage collector and supporting libraries for thread parallelism) [21] along with algebraic effects [22], with the objective for users to implement their own concurrency primitives. In recent time, several libraries implement their own flair of futures and promises, this time using a direct-style instead of the previous monadic one. Most of them, including the most developed library `eio` [15], and our own implementation, use a core API summarised in Figure 1.

```
1  type 'a Promise.t
2  (** Promises containing values of type ['a] *)
3
4  val Promise.create : unit -> 'a Promise.t * ('a -> unit)
5  (** [Promise.create ()] creates a promise explicitly and
       returns both the promise and the function to be called
        for its resolution *)
6
7  val Promise.async : (unit -> 'a) -> 'a Promise.t
8  (** [Promise.async (fun () -> e)] executes [e] in a
       promise *)
9
10 val Promise.get : 'a Promise.t -> 'a
11 (** [Promise.get p] makes a blocking read on promise [p]
       *)
12
13 val Promise.await : 'a Promise.t -> 'a
14 (** [Promise.get p] makes a non-blocking read on promise [
       p] *)
```

Fig. 1: A simple API for promises

The different elements of the API are commented in the figure. The type `Promise.t` is parameterised by its content (denoted by the type variable `'a`).

There are two ways to create a promise: `Promise.create` creates a promise and also returns the resolution function, while `Promise.async` associates a computation to the promise, actually creating a future. This library can be used in any setting, we thus differentiate between *non-blocking* operations such as `await`, which yield to another task, and *blocking* operations such as `get`, whose evaluation is stuck and blocks the current logical thread. It is then up to the invoker of this function and the scheduler to deal with this blocked state conveniently.

2.3 Promising Effects

Following [20], we now summarise a simplistic implementation of the `get` primitive for promises or futures using effects, as a way to introduce effects in the context of concurrency. As noted before, A promise, denoted here by the `promise` type is an atomic mutable box containing a status. The status is either `Resolved`, containing a value of type `'a`, or `Empty` waiting for a value.

```
type 'a status = Resolved of 'a | Empty
type 'a promise = 'a status Atomic.t
```

Using effects, Figure 2 showcases the implementation of blocking reads (`get` primitive) as explained below. On Line 1, we declare a new effect, called `Get`. From the usage perspective, an effect is a parameterised operation whose semantics is not specified, but whose typing is fixed: here, performing the `Get` effect takes as parameter a promise and returns the content. The `get` function, on Line 4, directly returns the value if the promise is fulfilled, or performs the `Get` effect otherwise. We still need to define what performing `Get` actually does. This is done via an *effect handler*, one Line 9. From the definition perspective, effects

```
1  effect Get : 'a promise -> 'a (** A new 'Get' effect *)
2
3  (** User-level function for blocking reads on promises *)
4  let get (p : 'a promise) : 'a = match Atomic.get p with
5    | Resolved v -> v
6    | Empty -> perform (Get p)
7
8  (** Underlying implementation of Get *)
9  let exec task = handle task() with
10   ...
11   | Promise.Get p, (k : 'a continuation) ->
12     let rec poll () = match Atomise.get p with
13       | Empty -> Domain.cpu_relax (); poll ()
14       | Resolved v -> continue k v
15     in poll ()
16   ...
```

Fig. 2: Function `Promise.get` and its effect handler

behave similarly to exceptions, except they allow resumption. The `exec` function executes a task in the context of a handler. When an effect is performed, it triggers the evaluation of the appropriate branch in the handler (here, Line 11) and binds the value contained in the effect (here, variable p is the promise to get). The handler also gives access to the *continuation* `k` at the point where the effect was performed. To implement `get`, we repeatedly poll the content of the promise until a value is obtained, and resume the continuation `k` with the value, thus resuming the execution of the task.

The continuation `k`, which is applied directly here, is in fact a first class value and can be passed around and stored. This allows implementing other operations on promises and other concurrency primitives, by defining a scheduler that manipulates continuations directly in user-land.

As indicated before, this implementation exactly mirrors (albeit with some simplifications) the ones in the current OCaml ecosystem. We now move on to a novel usage of algebraic effects by combining them with objects to implement active objects in OCaml.

3 An OCaml library for Active Objects

Our first contribution is an OCaml library, `actors-ocaml` available at https://github.com/Marsupilami1/actors-ocaml, which implements promises and active objects on top of OCaml's new features: effect handlers, to handle concurrency; and "domains", threads accompanied by their memory-managed heap, which acts as OCaml's parallelism units. We start by a showing our overlay for active objects before presenting its translation to effects.

3.1 Active objects

We showcase a first example of active object in OCaml in Figure 3. To create a new active object, we introduce a new dedicated syntax[3] `object%actor`[4]. It functions similarly to an OCaml object, with private fields, introduced by `val` and public `method`s. Here, we create an active object with one local field `x` initialised to 0, and three methods to `set` the local field, `get` it, and `multiply` it by a provided integer. Accessing private fields is transparent inside the active object, as if it was a normal variable, but forbidden outside the active object. We will see below that we use OCaml domains to implement such a local memory.

In OCaml, objects are typed structurally, with a type that reflects all their methods. Our active objects follow a similar trend: the object type is delimited by `< ... >` and contains a list of all methods. For instance, `get : int` (Line 3), indicates that the method `get` takes no argument and returns an integer.

[3] In our implementation we choose to adopt the *Actor* terminology instead of active objects because we believe actors are better known in the functional programming community.

[4] For this purpose, we use PPX, a specific hook that allow to extend OCaml with new lightweight syntax extensions

```
1 let a = object%actor
2   val mutable state = 0
3   method set n = state <- n
4   method get = state
5   method multiply n = state*n
6 end
```

(a) Object definition

```
1 val a : <
2   set : int -> unit;
3   get : int;
4   multiply : int -> int
5 > Actor.t
```

(b) Inferred type

```
1 a#.set 10;
2 let x : int Promise.t = a#!get in
3 let y : int Promise.t = a#!multiply (Promise.await x) in
4 let z : unit Promise.t = a#!set (Promise.await x) in
5 a#.get + Promise.get y
```

(c) Example of use

Fig. 3: A simple example using active object

`set : int -> unit` marks a method taking an integer and returning nothing. Note that the field is not shown in the type, since it represents internal state. This is crucial for active objects, as local fields are stored locally and shouldn't be accessed by other active objects. To distinguish active objects from normal objects, the structural type that consists of the methods is wrapped, giving the type `< .. > Actor.t`.

Actual usage of active objects is where we depart from traditional OCaml objects. Indeed, active objects support two types of method calls: `a#.get`, in Line 1, is *synchronous*. Such calls are either blocking if made externally, similarly to `Promise.get`, or direct if made internally by the actor itself. `a#!get` in Line 2 is *asynchronous*, which wraps the result in a Promise. `Promise.create` is called to create the promise and associates a dedicated resolver with the triggered call. The promise is returned to the invoker that can then perform `Promise.get` and `Promise.await` on it. The programmer cannot explicitly resolve the promise and can only access its value: promises returned by active objects are in fact *futures*, similarly to other active object languages.

3.2 Encapsulation and Data-race Freedom

Our library takes advantage of the OCaml type system to provide safe encapsulation of state and safe abstraction. Indeed, local variables, such as the `state` field in Figure 3, are hidden. Access can thus only be made inside methods. This ensures proper abstraction since only fields that are exposed through getters and setters can be accessed. It also ensures the absence of data-races, since methods are not executed concurrently (unless programmer explicitly use lower-level constructs, such as shared memory). Naturally, this is only true if two crucial

properties are ensured: mutable access cannot be captured, and it is impossible to return mutable values shared with the internal state.

Capture Methods calls in OCaml are currified by default. For instance `a#! multiply` returns a closure of type `int -> (int Promise.t)` encapsulating message sending to `a` and retrieval of a result from `a`. Furthermore, functions are first class, and can be returned by methods. While this provides great integration into the rest of the language, this means that we need to be particularly careful with captures in methods. We illustrate this in Figure 4, with an incorrect implementation of the `multiply` method. Here, we return a closure capturing an access to the internal field `state`. Such closure should never be executed in the context of another active object. We detect such ill-conceived code and return the error shown below, instructing the user to first access the state before capturing the value.

In theory, this is a simple matter of name resolution. In practice, name resolution in OCaml is complex, and relies on typing information which can't be accessed by syntax extensions such as the one we develop. We implement a conservative approximation.

```
let a = object%actor
  val mutable state = 0
  method multiply = let f n = state * n in f
end
```

(a) An incorrect implementation of `multiply`

```
Closures cannot capture internal mutable state, you may
    want to use something like:
|
| let a = state in fun _ -> ... a ...
|
Instead of:
|
| fun _ -> ... state ...
|
```

(b) The error message for an illegal capture

Fig. 4: An example of illegal capture and its error message

Mutability and sharing Code that respects the criterion mentioned above can still exhibit data-races, for instance by returning the content of a field which manifest internal mutability, such as arrays. Preventing such mistakes is a bit more delicate: with the strong abstraction of OCaml, the implementation of a

data-structure can be completely hidden, and hence its potential mutability. A static type analysis is therefore insufficient. A dynamic analysis of the value is similarly insufficient (mutable and immutable records are represented similarly in OCaml). The last common solution to this problem, to make a deep copy of returned values, is costly both in terms of time and loss of sharing.

So far, we opted to only support immutable values in fields, and do not provide any guarantees when mutable values are used. Thankfully, immutable values are the default in OCaml and are largely promoted for most use-cases. In the future, we plan to combine static and dynamic analysis to inform where to insert deep copies.

3.3 Active Object Desugaring

We now have all the ingredients to explain how the OCaml code for the active object is generated from the programmer's input. An example of such translation is given in Figure 5. The first important notion is to use memory local to the domain to store the internal fields. Using domains, this is done via the `DLS` (for Domain Local Storage, analogous to thread local storage), see for instance line 2

```
1 let a = object%actor
2   val mutable state = 0
3   method set n = state <- n
4   method get = state
5 end
```

⇓

```
1 let a = object (self)
2   val __state = DLS.new_key (fun _ -> 0)
3   method __meth_set n = DLS.set __state n
4   method set n =
5     let p, resolver = Promise.create () in
6     Actor.send self
7       (Scheduler.process resolver
8         (fun _ -> self#__meth_set n)) ;
9     p
10  method __meth_get = DLS.get __state
11  method get =
12    let p, resolver = Promise.create () in
13    Actor.send self
14      (Scheduler.process resolver
15        (fun _ -> self#__meth_get)) ;
16    p
17  end
```

Fig. 5: Simple active object code (top) and its translation (bottom)

```
1 object%actor (self)
2   method syracuse n =
3     if n = 1 then 1
4     else
5       let next = if n mod 2 = 0 then n/2 else 3*n+1 in
6       self#!!syracuse next
7 end
```

Fig. 6: Simple use of delegation

of the translated code. All reads and writes are then replaced by `DLS` functions. The second transformation aims to separate method calls (i.e., message sent), and execution, and can be observed on line 3 and 4: Each method is split in two. The first *hidden* method, shown on line 3, contains the computational content. The second is the actual entry point: it proceeds by creating a promise; launch a new task; and return the promise. The goal of the task is to queue a message in the actor's mailbox, via `Actor.send`, and then launch a process which eventually resolves the promise; this is done by `Scheduler.process`.

3.4 Forward

While handling a method, one might want to delegate the computation to another active object or method. With traditional asynchronous calls such as `#!` or `await`, this would involve unwrapping and rewrapping the promises. Dealing efficiently with delegation in asynchronous invocations is a well-studied problem [6, 7, 5]. In [6], one construct called *forward* was suggested for such delegations; it was then shown that directly forwarding an asynchronous invocation (`return(async(e))`) could be efficiently and safely implemented using promises.

We can easily adapt this approach to our actors. We also implement delegation calls by syntactically identifying such optimisable situation with the primitive: `actor#!!m`. Figure 6 illustrates a simple program using such method delegation; the statement `self#!!syracuse next` delegates the current invocation to another one. These calls act as `return` in many languages, and ignore any computations that would come after in the method. From the functional programming point of view, this is analogous to tail-calls. Tail-calls exploit synchronous calls in return positions to eschew using additional stack space. Forward statement exploits asynchronous calls in return position to eschew indirection of promises.

In a more general case, we can simply forward[5] a promise as the future resolution of the current promise. A statement similar to the one of Encore, `Actors.forward p` performs such a shortcut where `p` is a `'a Promise.t`.

We implement the two forwarding constructs presented above as *effects* in the library. Similarly to capturing issues highlighted in previous sections, delegation

[5] In the future, we hope to turn asynchronous calls in a `forward` into delegation automatically.

calls should not be captured in a closure: indeed, it wouldn't be clear which indirection to avoid[6]. We forbid such situations (dynamically, via a runtime test).

3.5 Runtime Support

From a parallelism point of view, we rely on domains, which are threads equipped with a private heap and a garbage collector. There is also a global, shared heap. In practice, we spawn a pool of domains at the start of the execution. This pool of domains is fixed for the whole execution. Similarly to many other implementations, multiple actors may share a domain, and will use cooperative scheduling together.

Cooperative scheduling is implemented using effects and continuations, similarly to the one implemented in the introduction. To make this scheduler more realistic and fair, we implement the following optimisations:

- Each domain contains a first round-robin scheduler in charge of scheduling between active objects hosted on the same domain. Spawning of new actors is implemented at this layer, enabling the choice of an arbitrary domain to spawn it. Synchronous method calls in the same domain are transparently turned into direct calls (instead of asynchronous calls followed by a synchronisation when the domain is different).
- Each active object contains an OCaml object with the methods of the object, as described above, and a second round-robin scheduler which schedules the promises currently executed by this actor. Instead of a traditional mailbox of messages, active objects contain a queue of thunks to be executed. In the case of method calls, each thunk contains a call to the underlying OCaml object as a closure. Forwards and delegation calls are implemented at this second layer, which is aware of all the details pertaining to the actor.
- Unresolved promises contain a list of *callbacks*, i.e., other promises that are currently waiting on it. This allows the implementation of passive waits for unresolved promise reads.

Note that this implies we have *two* effects handlers, both providing slightly differing implementation of the base effects related to promises (`Async`, `Get`, `Await`). Indeed, promises can appear outside of actors, but should be handled locally if they appear inside one.

4 Future and Effect λ-calculi

The rest of this article is dedicated to the formal description of the compilation of Futures to Effects. For this purpose, we first introduce our protagonists: A common imperative base (Section 4.1), the source future calculus (Section 4.2) often characterised in green, and the target effect calculus (Section 4.3) often characterised in blue. For all these calculi, we define small-step operational semantics in the sequential and parallel cases.

[6] Already in [6], the authors prevented forward from appearing inside a closure.

4.1 A Functional-Imperative Base

We define a standard λ-calculus with imperative operations that will be the base language for our other definitions and semantics. The syntax is given in Figure 7. As meta-syntactic notations, we use overbar for lists ($\overline{e}$ a list of expressions) and brackets for association maps ($\boxed{\ell \mapsto e}$). $\mathrm{Dom}(M)$ is the domain of M and $\emptyset$ is the empty map. $M\,[v \mapsto v']$ is a copy of M where v is associated to v', $M \setminus v$ is a copy of M where v is not mapped to anything ($v \notin \mathrm{Dom}(M \setminus v)$).

Most expression and values are classical. The substitution of x by e' in e is denoted $e\,[x \leftarrow e']$. Stores are maps indexed by location references, denoted ℓ. **Id** denotes unique identifiers that can be crafted during execution, which will be useful in our two main calculi. Location references and identifiers should not occur in the source programs and only appear during evaluation. We also define evaluation contexts C that are expressions with a single hole $\square$. Evaluation contexts are used in the semantics to specify the point of evaluation in every term, ensuring a left-to-right call-by-value evaluation. We classically rely on evaluation contexts, $C[e]$ is the expression made of the context C where the hole is filled with expression e. Figure 8 defines a semantics for this base calculus; it is similar to what can be found in the literature. It expresses a reduction relation, denoted $\longrightarrow$, of pairs store×expression.

Important note The rules of Figure 8 act on the syntax of imperative λ-calculus. However, in the next section we will re-use $\longrightarrow$ on terms of bigger languages, with the natural embedding that $\longrightarrow$ rules only are able to handle the λ-calculus

$$
\begin{array}{lll}
e ::= v & \text{(Values)} & v ::= \ell \in \mathbf{Loc} \quad \text{(References)} \\
\quad \mid x \in \mathbf{Var} & \text{(Variables)} & \quad \mid \lambda x.e \quad \text{(Functions)} \\
\quad \mid () & \text{(Unit)} & \quad \mid i \in \mathbf{Id} \quad \text{(Identifiers)} \\
\quad \mid \lambda x.e \mid (e_1\ e_2) & \text{(Functions)} & \quad \mid c \in \mathbf{Const} \quad \text{(Constants)} \\
\quad \mid \texttt{newref}(e) \mid !x \mid x := e & \text{(References)} & \sigma ::= \boxed{\ell \mapsto v} \quad \text{(Store)}
\end{array}
$$

$$
C ::= \square \mid (C\ e) \mid (v\ C) \mid \texttt{newref}(C) \mid C := e \mid \ell := C \qquad \text{(Eval. context)}
$$

Fig. 7: Syntax for the base impure λ-calculus

$$
\frac{\ell \notin \mathrm{Dom}(\sigma)}{\sigma, \texttt{newref}(v) \longrightarrow \sigma\,[\ell \mapsto v]\,, \ell} \qquad \frac{(\ell \mapsto v) \in \sigma}{\sigma, !\ell \longrightarrow \sigma, v} \qquad \frac{}{\sigma, (\ell := v) \longrightarrow \sigma\,[\ell \mapsto v]\,, ()}
$$

$$
\frac{\sigma, e \longrightarrow \sigma', e'}{\sigma, C[e] \longrightarrow \sigma', C[e']} \qquad \frac{}{\sigma, (\lambda x.e\ v) \longrightarrow \sigma, e\,[x \leftarrow v]}
$$

Fig. 8: Semantics for the base impure λ-calculus

primitives but will manipulate terms and reduction contexts of the other languages. The alternative would be to define from the beginning the syntax and reduction contexts of our language as the largest syntax including all the three considered languages (λ-calculus, FUT, and EFF). We chose here to adopt a more progressive presentation despite the slight abuse of notation this involves on the formal side.

In the rest of this article, we also assume additional constructs which can be classically encoded in the impure λ-calculus:

- Let-declaration: `let x = ... in ...`
- Sequence: `e; e'`
- Mutually recursive declarations: `let rec ... and ...`
- Mutable maps indexed by values: empty map $\{\}$, reads $M[e]$, writes $M[e] \leftarrow e'$, and deletions `del` $M[e]$
- Pattern matching on simple values: `match ... with ...`

4.2 Futures and Cooperative Scheduling

Our λ-calculus with futures shares some similarities with the concurrent λ-calculus with futures [18], but without future handlers or explicit future name creation and scoping, resulting in a simpler calculus. Our calculus can also be compared to the one of Fernandez-Reyes et al. [6] but with cooperative scheduling with multiple threads, and imperative aspects.

The λ-calculus of previous section is extended as shown in Figure 9. Four new constructs are added to the syntax: `spawn()` spawns a new processing unit; `asyncAt`(e, e') starts a new task e in the processing unit e' and creates a future identifier f, when the task finishes, this *resolves* the future f; `get`(e), provided e is a future identifier, blocks the current processing unit until the future in e is resolved; `await`(e) is similar but releases the current processing unit until the future is resolved. Evaluation contexts are trivially extended.

As shown in Figure 9, we suppose that *future identifiers* have a specific shape of the form $fut = (tid, lf)$ where tid is a thread identifier and lf is a local future identifier. Tasks map expressions to future identifiers, when the expression is fully evaluated (to a value) the future is *resolved*.

The dynamic syntax is expressed in two additional layers: above the λ-calculus layer of Figure 8, Figure 10 expresses the reduction relation in a given processing unit, and Figure 11 extends this local semantics to a parallel semantics with several processing units.

The local semantics in Figure 10 is based on configurations of the form σ, F, s where σ is a shared mutable store, F is the map of futures, and s is a state. If the expression in the current task is fully evaluated to a value, the task is finished, the future is resolved and put back into the task list, the state of the processing unit is `Idle` (rule RETURN). Rule STEP performs a λ-calculus step (see Figure 8). `get`(f) can only progress if the future f has been resolved, in which case the value associated with the future is fetched (rule GET). There are two rules for `await`(f): if the future is resolved `await`(f) behaves like `get`(f); if it is not

$$
\begin{array}{rll}
e ::= & \ldots & \text{(Base language)} \\
 & \mid \texttt{asyncAt}(e, e) & \text{(Creation)} \\
 & \mid \texttt{get}(e) & \text{(Blocking read)} \\
 & \mid \texttt{await}(e) & \text{(Non-blocking read)} \\
 & \mid \texttt{spawn}() & \text{(Spawn process)} \\
C ::= & \cdots \mid \texttt{asyncAt}(C, e) \mid \texttt{asyncAt}(v, C) & \\
 & \mid \texttt{await}(C) \mid \texttt{get}(C) & \text{(Evaluation Contexts)}
\end{array}
$$

$$
\begin{array}{rll}
tid \in \mathbf{ThreadId} \subset \mathbf{Id} & & \\
lf \in \mathbf{LocalFutures} \subset \mathbf{Id} & & \\
f ::= (tid, lf) \in \mathbf{Id} & & \text{(Future Ids)} \\
F ::= \boxed{f \mapsto e} & & \text{(Tasks)} \\
s ::= \texttt{Idle} \mid (f \mapsto e) & & \text{(Exec. State)} \\
P ::= \|_{i \in I}\, s^i & & \text{(Parallel exec. state)} \\
 & & I \subseteq \mathbf{ThreadId}
\end{array}
$$

Fig. 9: Syntax for the FUT language

$$
\textsc{step}\ \frac{\sigma, e \longrightarrow \sigma', e'}{\sigma, F, (f \mapsto e) \longrightarrow \sigma', F, (f' \mapsto e')}
\qquad
\textsc{get}\ \frac{(f' \mapsto v) \in F}{\sigma, F, (f \mapsto C[\texttt{get}(f')]) \longrightarrow \sigma, F, (f \mapsto C[v])}
$$

$$
\textsc{await-val}\ \frac{(f' \mapsto v) \in F}{\sigma, F, (f \mapsto C[\texttt{await}(f')]) \longrightarrow \sigma, F, (f \mapsto C[v])}
\qquad
\textsc{await-yield}\ \frac{\nexists v.\, (f' \mapsto v) \in F}{\sigma, F, (f \mapsto C[\texttt{await}(f')]) \longrightarrow \sigma, F\left[f \mapsto C[\texttt{await}(f')]\right], \texttt{Idle}}
$$

$$
\textsc{return}\ \frac{}{\sigma, F, (f \mapsto v) \longrightarrow \sigma, F\left[f \mapsto v\right], \texttt{Idle}}
$$

$$
\textsc{async}\ \frac{f' = (tid, lf) \qquad f' \notin \mathrm{Dom}(F)}{\sigma, F, (f \mapsto C[\texttt{asyncAt}(e, tid)]) \longrightarrow \sigma, F\left[f' \mapsto e\right], (f \mapsto C[f'])}
$$

Fig. 10: Semantics for FUT — $\sigma, F, s \longrightarrow \sigma, F, s$

resolved the task is interrupted (it returns to the task pool), the processing unit becomes `Idle`. Finally, rule ASYNC starts a new task: the effect of $\texttt{asyncAt}(e, tid)$ is first to forge a future identifier containing the thread identifier tid and another identifier lf so that the pair (tid, lf) is fresh, a task is created, associating e to the new future.

The management of processing units and thread identifiers is the purpose of the parallel semantics in Figure 11. It expresses the evaluation of configurations of the form σ, F, P where P is a parallel composition of processing units. $P \parallel s^i$ is used both to extract the execution state of thread i form the parallel composition P and to add it back. Rule ONE-STEP simply triggers a rule of the local semantics in Figure 11. Rule SPAWN spawns a new thread, creating a fresh thread identifier that will be used in an `AsyncAt` statement to initiate work on this thread (the new thread is initially `Idle`). Finally, if s^i is `Idle`, no task is currently running and a new task can be started on the processing unit i by the rule SCHEDULE.

$$\textsc{one-step}\quad \frac{\sigma, F, s \longrightarrow \sigma_2, F_2, s_2}{\sigma, F, P \parallel s^i \longrightarrow_{||} \sigma_2, F_2, P \parallel {s_2}^i}$$

$$\textsc{spawn}\quad \frac{tid \notin tids(P) \cup \{i\}}{\sigma, F, P \parallel (f \mapsto C[\texttt{spawn}()])^i \longrightarrow_{||} \sigma, F, P \parallel (f \mapsto C[tid])^i \parallel \texttt{Idle}^{tid}}$$

$$\textsc{schedule}\quad \frac{((i, lf) \mapsto e) \in F \qquad e \text{ is not a value}}{\sigma, F, P \parallel \texttt{Idle}^i \longrightarrow_{||} \sigma, F \setminus (i, lf), P \parallel ((i, lf) \mapsto e)^i}$$

Fig. 11: Parallel semantics for Fut — $\sigma, F, \|_{i \in I}\, s^i \longrightarrow_{||} \sigma, F, \|_{i \in I}\, s^i$

An initial configuration for an Fut program e_p consists of the program associated with a fresh task identifier i and a fresh future identifier f, with an empty store and future map: $\emptyset, \emptyset, (f \to e_p)^i$

4.3 Effects

We now extend the base calculus of Section 4.1 with effects. For the moment this extension is independent of the previous one, they are used separately in this article even though composing the two extensions would be perfectly possible. Indeed, we transform programs with only futures into programs with only effects but having a language with at the same time futures and effects would also make sense.

Figure 12 shows the syntax of the parallel and imperative λ-calculus with effects. Parallelism is obtained by the keyword $\texttt{spawn}(e)$ that creates a new thread in the same spirit as in the previous section. $\texttt{handle}(e)\{h\}$ runs the expression e under the handler h, if an effect is thrown by $\texttt{throw}(E(C))$ inside e, and if h can handle this effect, the handler is triggered. Rule HANDLE-EFFECT in Figure 13 specifies the semantics of effect handling. Suppose an effect E is thrown, the first encompassing handler that can handle this effect is triggered: if a rule $(E(x), k \mapsto e)$ is in the handler, then the handler e is triggered with x assigned to the effect value v and k assigned to the continuation of the expression that triggered the effect. The interplay between evaluation contexts and the captured_effects() function captures the closest matching effect. Rule HANDLE-STEP handles the case where the term e performs a reduction not related to effect handling. If e finally returns a value, Finally, rule HANDLE-RETURN deals with the case where the handled expression can be fully evaluated without throwing an effect; it triggers the expression corresponding to the success case $x \mapsto e$ in the handler definition. Note that we don't reinstall the handler after triggering the rule, corresponding to the *shallow* interpretation of effect handlers [12].

Figure 14 shows the parallel semantics of effects. The only specific rule is SPAWN, which spawns a new thread with a fresh identifier. Note that in Eff, the parameter of `spawn` is the expression to be evaluated in the new thread, with its own thread identifier as argument.

$$
\begin{array}{rll}
e ::= \ldots & & E \in \textbf{Symbol} \\
\quad \mid \texttt{handle}(e)\{h\} \mid \texttt{throw}(E(e)) & & k \in \textbf{Var} \\
\quad \mid \texttt{spawn}(e) & & h ::= \left[\overline{E(x), k \mapsto e};\ x \mapsto e\right] \\
C ::= \cdots \mid \texttt{handle}(C)\{h\} \mid \texttt{throw}(E(C)) & & \\
\quad \mid \texttt{spawn}(e) \qquad \text{(Evaluation Contexts)} & &
\end{array}
$$

Fig. 12: Eff Syntax

$$
\textsc{handle-step}\ \frac{\sigma, e \longrightarrow \sigma', e'}{\sigma, e \longrightarrow \sigma', e'} \qquad \textsc{handle-return}\ \frac{(x \mapsto e) \in h}{\sigma, \texttt{handle}(v)\{h\} \longrightarrow \sigma, e\,[x \leftarrow v]}
$$

$$
\textsc{handle-effect}\ \frac{(E(x), k \mapsto e) \in h \qquad E \notin \text{captured\_effects}(C)}{\sigma, \texttt{handle}(C[\texttt{throw}(E(v))])\{h\} \longrightarrow \sigma, e\,[x \leftarrow v]\,[k \leftarrow \lambda y.C[y]]}
$$

$$
\begin{array}{rl}
\text{captured\_effects}(\square) & = \emptyset \\
\text{captured\_effects}(\texttt{handle}(C)\{h\}) & = \text{captured\_effects}(C) \cup \{E \mid (E(x), k \mapsto e) \in h\} \\
\text{captured\_effects}(\ldots) & = \ldots \quad \text{(by immediate recursion otherwise)}
\end{array}
$$

Fig. 13: Semantics for Eff — $\sigma, e \longrightarrow \sigma, e$

$$
\textsc{seq}\ \frac{\sigma, e \longrightarrow \sigma', e'}{\sigma, P \parallel e^i \longrightarrow_{||} \sigma', P \parallel e'^i} \qquad \textsc{spawn}\ \frac{tid \notin tids(P) \cup \{i\}}{\sigma, P \parallel C[\texttt{spawn}(e)]^i \longrightarrow_{||} \sigma, P \parallel C[tid]^i \parallel (e\ tid)^{tid}}
$$

Fig. 14: Parallel semantics for Eff — $\sigma, \|_{i \in I}\, e^i \longrightarrow \sigma, \|_{i \in I}\, e^i$

An initial configuration for an Eff program e_p simply consists of the program associated with a fresh task identifier i and with an empty store: $\emptyset, e_p^i$.

5 Compilation of futures into effects

In this section we define a transformation from Fut to Eff that translates from our concurrent λ-calculus with futures into the calculus with effect handlers. We then prove its correctness.

5.1 Translating Fut into Eff

Figure 15 shows the translation $[\![e]\!]_p$ that transforms a Fut program e into an Eff program with the same semantics. The color highlighting in the definition can be ignored at first. It will be used in the proof in the next section. $[\![e]\!]_p$ is

$$[\![e]\!]_p \triangleq \texttt{let } tasks = \{\} \texttt{ in}\ \texttt{let rec } poll(fut) = Poll \texttt{ in}\ \texttt{let rec } continue(fut, k, t) = Continue\ \texttt{and } run(t) = Run \texttt{ in}\ continue(fresh(), \lambda().\,[\![e]\!]_e, t_0)$$

$$[\![\texttt{asyncAt}(e,t)]\!]_e = \texttt{throw}(Async(\lambda().\ [\![e]\!]_e, [\![t]\!]_e))$$
$$[\![\texttt{await}(e)]\!]_e = \texttt{throw}(Await([\![e]\!]_e))$$
$$[\![\texttt{get}(e)]\!]_e = \texttt{throw}(Get([\![e]\!]_e))$$
$$[\![\texttt{spawn}()]\!]_e = \texttt{throw}(Spawn())$$
$$[\![x]\!]_e = x \qquad [\![v]\!]_e = v$$
$$[\![e]\!]_e = \ldots \text{ (immediate recursion otherwise)}$$

Where

$Continue \triangleq$

```
handle( k() ){
| x ↦
    tasks[fut] ← 𝕍(x);
    run(t)
| Async(job, t'), k' ↦
    let fut' = (t', fresh()) in
    tasks[fut'] ← ℂ(job);
    continue(fut, λ().k'(fut'), t)
| Await(fut_a), k' ↦
    match tasks[fut_a]{
      | 𝕍(v) ↦ continue(fut, λ().k'(v))
      | _ ↦ let k''() = k'(throw(Await(fut_a))) in
            tasks[fut] ← ℂ(k'');
            run(t)
    };
| Spawn(), k' ↦
    let t' = spawn(run) in
    continue(fut, λ().k'(t'), t)
| Get(fut_g), k' ↦
    let v = poll(fut_g) in
    continue(fut, λ().k'(v), t)
}
```

$Run \triangleq$

```
let (fut, k) =
  pop(tasks, t)
in
tasks[fut] ← None;
continue(fut, k, t)
```

$Poll \triangleq$

```
match tasks[fut]{
  | 𝕍(v) ↦ v
  | _ ↦ poll(fut)
}
```

Pop

$$\frac{fut = (tid, lf) \qquad tasks[fut] = \mathbb{C}(k)}{pop(tasks, tid) \longrightarrow (fut, k)}$$

Fig. 15: Translation from Fut to Eff

the top level **p**rogram transformation while $[\![e]\!]_e$ is used to compile **e**xpression; this transformation simply replaces FUT specific expressions into expressions throwing an effect with adequate name and parameters. The handling of effects is defined at the top level, i.e. when translating the source program.

$[\![e]\!]_p$ creates a program that uses a pool of tasks called *tasks* and three functions that manipulate it. *tasks* is implemented by a mutable map from future identifiers to tasks, which can be of two kinds: continuations of the form $\mathbb{C}(k)$ or values of the form $\mathbb{V}(v)$.

The main function is *continue*, it sets up a handler dealing with all the effects of FUT. It first evaluates the thunk continuation parameter k. Then it reacts to the different possible effects as follows. The first branch describes the behavior when $k()$ throws no effect and simply returns a value. In this case, the task is saved as a value $\mathbb{V}(v)$ (the future is resolved). The *Async* effect adds a new task to the task pool and *continues* the execution of the current task with the continuation k' and the newly created future fut'. The *Await* effects checks whether the future fut_a in the task pool has been resolved or not; if it is resolved the task continues with the future value, otherwise the task is put back in the pool of tasks (keeping the *Await* effect at the head of the continuation). The *Get* effect is similar to the resolved case of *Await* but does not allow the task to be returned to the pool of tasks. Instead, if the future is not resolved the thread actively polls the matching task until the future is finally resolved using the auxiliary *poll* function. The *Spawn* effect case spawns a new thread that runs the *run* function. In each case where the task does not continue, the body of the function *run* is triggered.

The function $run(t)$ uses the external function $pop(tasks, t)$ to fetch a new unresolved task that should run on thread t, the task is thus of the form $\mathbb{C}(k)$ and the thread continues by evaluating the thunk continuation k.

5.2 Correctness of the Compilation of Actors into Effects

We define in this section a hiding semantics and will prove strong bisimulation between the source program and the hiding semantics of the transformed program.

5.2.1 Hiding Semantics In translation such as the one defined here, the compiled program must often take several more "administrative" steps than the source program. This makes proof by bisimulation more complex, and requires using weak bisimulation that ignores some steps marked as internal.

In this article we take a stronger approach and prove strong bisimilarity on a derived transition relation. The principle is that internal steps of the transformed program are called silent, and they are by nature deterministic and terminating. We can thus consider that we "normalise" the runtime configuration of the transformed program by systematically applying as many internal steps as possible until a stable state is reached. We discuss this idea further in Section 6.

We first state that $hidden(e)$ is true if the top level node in the syntax of e is *colored*; where colored means the term is surrounded by a colored box: $\boxed{e}$. There should be no ambiguity on the node of the syntax that is colored (at least in our translation).

Definition 1 (Hiding semantics). *We define a hiding operation to hide parts of the reduction. It works as follows. We can define a h-reduction* $\longrightarrow_{\mathrm{h}}$ *that puts a* τ *label on the transitions that target a node of the syntax that is hidden:*

$$\frac{\sigma, e \longrightarrow_{||} \sigma', e' \qquad hidden(e)}{\sigma, e \xrightarrow{\tau}_{\mathrm{h}} \sigma', e'} \qquad \frac{\sigma, e \longrightarrow_{||} \sigma', e' \qquad \neg hidden(e)}{\sigma, e \longrightarrow_{\mathrm{h}} \sigma', e'}$$

We finally define the hiding semantics as one non-hidden step followed by any number of hidden step, until no further hidden step can be performed[7]*:*

$$\sigma, e \underset{||}{\Longrightarrow} \sigma, e \iff \sigma, e \longrightarrow_{\mathrm{h}} \xrightarrow{\tau}_{\mathrm{h}}{}^{*} \sigma', e' \not\xrightarrow{\tau}_{\mathrm{h}}$$

Note that, considering the nodes colored in our translation, the transitions marked as τ should only have a local and deterministic effect on the program state. In practice there are some hidden statements that spawn a thread or launches task for example, but they are immediately and deterministically preceded by a decision point that is visible, here the reaction to an effect. The interleaving of the tau transition have no visible effect on the global state, only the state along the visible transitions is important. This property will be made explicit in our proof of correctness. As a consequence, because the hidden step commutes with all the other steps, each execution of a Fut program compiled into Eff can be seen as a succession of $\underset{||}{\Longrightarrow}$. Additionally, except when polling futures the transitive closure of hidden steps terminate. We have the following property, relating our middle-step and small-step semantics.

Theorem 1 (Middle-step semantics). *Consider* $e_1 = [\![e_f]\!]_p$*. Any Eff reduction of* e_1 *can be seen as a hiding semantics reduction, modulo a few hidden steps, and a few get operations on unresolved futures:*

$$\sigma_1, e_1 \longrightarrow_{||}{}^{*} \sigma_2, e_2 \implies \exists \sigma_3, e_3, \sigma_4, e_4. \bigwedge \begin{array}{l} \sigma_1, e_1 \underset{||}{\Longrightarrow}{}^{*} \sigma_3, e_3 \\ \sigma_2, e_2 \xrightarrow{\tau}_{\mathrm{h}}{}^{*} \sigma_4, e_4 \\ \sigma_3, e_3 \xrightarrow{\text{handle-get}}_{||}{}^{*} \sigma_4, e_4 \end{array}$$

Where $\sigma_3, e_3 \xrightarrow{\text{handle-get}}_{||}{}^{*} \sigma_4, e_4$ *is application (inside an appropriate context) of a* HANDLE-EFFECT *rule with a* Get *effect on an unresolved future. in particular, if all futures are resolved,* $\sigma_3, e_3 = \sigma_4, e_4$*.*

This theorem is true because the hidden semantic steps commute, only a special case is needed for handling the polling of unresolved futures.

[7] $\longrightarrow^{*}$ denotes the reflexive transitive closure of the relation $\longrightarrow$.

5.2.2 Bisimulation Definition To help with our bisimulation definition, we now define a few execution contexts that appear commonly in the proof. C_{rec} is the context that corresponds to the recursive knot introduced by `let rec`. Indeed, since `let rec` expresses recursion as an encoding into λ-calculus, the encoding will appear again in each task and can be sugared/de-sugared at will. In addition, C_c and C_r are the contexts in the translated program where *continue* and *run* are respectively executed, parameterised by all their free variables. In the following we thus start each task by C_{rec}, C_c or C_r. More precisely:

$$C_{rec}[\ell_{threads}] \triangleq \left(\begin{array}{l} \texttt{let rec } \mathit{poll}(\mathit{fut}) = \mathit{Poll} \texttt{ in} \\ \texttt{let rec } \mathit{continue}(\mathit{fut}, k, t) = \mathit{Continue} \\ \texttt{and } \mathit{run}(t) = \mathit{Run} \texttt{ in} \\ \square \end{array} \right) [\mathit{tasks} \leftarrow \ell_{threads}]$$

$$C_c[\ell_{threads}, fut, K', t] \triangleq C_{rec}[\ell_{threads}][\mathit{continue}(fut, k, t)\,[k() \leftarrow K']]$$

$$C_r[\ell_{threads}, t] \triangleq C_{rec}[\ell_{threads}][\mathit{run}(t)]$$

Definition 2 (Relation over configurations). *Let* R *be a relation over pairs of a* Fut *configuration* C_{Fut} *and a* Eff *configuration* C_{Eff}*. We also note* R_e *a relation over pairs of configuration states in* Fut *(i.e.,* $(\sigma, \ell_{threads})$*) and in* Eff *(i.e.,* (σ, F)*).*

Figure 16 defines both relations. The purpose of the relation is to prove the correctness of our compilation scheme. We will prove that R *is a strong bisimulation.* R *is indexed either by* $\|$ *for parallel configurations, and by a given* t *to reason about single-threaded configurations of thread* t*. For single-threaded configurations, the computation can either be in the* CONTINUE *case, or the* RUN *case. The most complex relation is on the environments, which details the content of the* $\ell_{threads}$ *values.*

The translation $[\![]\!]_e$ can straightforwardly be extended to contexts (where $[\![\square]\!]_e = \square$). Consequently, we have the following property:

Lemma 1 (Context compilation). $[\![C[e]]\!]_e \equiv [\![C]\!]_e\,[[\![e]\!]_e]$

Proof. By case analysis on the translation rules (and on contexts). □

5.2.3 Correctness of the compilation scheme We now establish the correctness of our translation by proving that the relation we exhibited in the previous section is a bisimulation.

Theorem 2 (Correctness of the compilation scheme). *The relation* $\mathrm{R}_{\|}$ *is a strong bisimulation where the transition on the* Eff *side is the hiding transition relation, and the transition on the* Fut *side is* $\longrightarrow_{||}$*. Formally, for all*

$$\textsc{env}\quad \frac{\begin{array}{c} F_e = F_{e,1} \uplus F_{e,2} \\ \forall f \in \mathrm{Dom}(F_v).\, F_v(f) \text{ is a value} \qquad \forall f \in \mathrm{Dom}(F_e).\, F_e(f) \text{ is not a value} \\ T_{e,1} = \left[f' \mapsto \mathbb{C}(\lambda().\, [\![e]\!]_e) \mid F_{e,1}(f') = e\right] \\ T_{e,2} = \left[f' \mapsto \mathbb{C}\,(\lambda().((\lambda x.C[x])\ e)) \mid [\![F_{e,2}(f')]\!]_e = C[e]\right] \\ T_v = \left[f' \mapsto \mathbb{V}([\![v]\!]_e) \mid F_v(f') = v\right] \end{array}}{\sigma_{base} \cup \{\ell_{threads} \mapsto T_{e,1} \uplus T_{e,2} \uplus T_v\}, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_{base}, F_e \uplus F_v}$$

$$\textsc{continue}\quad \frac{\sigma, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma', F}{\sigma, C_c[\ell_{threads}, f, [\![e]\!]_e, t]\ \ \mathrm{R}_t\ \ \sigma', F, (f \to e)} \qquad \textsc{run}\quad \frac{\sigma, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma', F}{\sigma, C_r[\ell_{threads}, t]\ \ \mathrm{R}_t\ \ \sigma', F, \texttt{Idle}}$$

$$\textsc{par}\quad \frac{\forall t \in T.\ \ \sigma, e_t\ \ \mathrm{R}_t\ \ \sigma', F, s_t}{\sigma, \|_{t \in T}(e_t)^t\ \ \mathrm{R}_\|\ \ \sigma', F, \|_{t \in T}(s_t)^t}$$

Fig. 16: Relation between Fut terms and their compiled version

configurations the following holds:

$$\sigma_1, P_1\ \ \mathrm{R}_\|\ \ \sigma_1', F_1, P_1'\ \wedge\ \sigma_1, P_1 \overset{\Longrightarrow}{||}\ \sigma_2, P_2 \implies \exists \sigma_2', F_2, P_2'.\quad \sigma_1', F_1, P_1' \overset{}{\longrightarrow_{||}}\ \sigma_2', F_2, P_2'\ \wedge\ \sigma_2, P_2\ \ \mathrm{R}_\|\ \ \sigma_2', F_2, P_2'$$

and

$$\sigma_1, P_1\ \ \mathrm{R}_\|\ \ \sigma_1', F_1, P_1' \wedge \sigma_1', F_1, P_1' \overset{}{\longrightarrow_{||}}\ \sigma_2', F_2, P_2' \implies \exists \sigma_2, P_2.\quad \sigma_1, P_1 \overset{\Longrightarrow}{||}\ \sigma_2, P_2 \wedge \sigma_2, P_2\ \ \mathrm{R}_\|\ \ \sigma_2', F_2, P_2'$$

so that for any Fut program p the initial configuration of the program and of its effect translation are bisimilar (with t_0 fresh, and f_0 is the fresh future identifier that has been chosen when triggering the first continue function.).

$$\emptyset, ([\![e_p]\!]_p\,[fresh() \leftarrow f_0])^{t_0}\ \ \mathrm{R}_\|\ \ \emptyset, \emptyset, (f_0 \mapsto e_p)^{t_0}$$

Proof (sketch). The proof of bisimulation follows a standard structure. For each pair of related configurations we show that the possible reductions made by one configuration can be simulated by the equivalent configuration (in the other calculus). Then a case analysis is performed depending on the rule applied. The set of rules is different between Fut and Eff calculi but on the Eff side, we need to distinguish cases based on the name of the triggered effect, leading to a proof structure similar to the different rules of Fut. Appendix A details the proof that the compiled program simulates the original one. By case analysis on the rule that makes the relation true and the involved reduction. This leads to seven different main cases; we prove simulation in each case. □

Finally, Theorems 1 and 2 allow us to conclude regarding the correctness of our compilation scheme. Indeed, each execution of a compiled program is equivalent to a middle-step reduction that itself simulates one of the possible executions of our Fut program. Conversely, any execution of our Fut program corresponds (modulo polling of unresolved futures) to a middle-step execution of its compilation, which is in fact one of the Eff executions of the compiled program.

6 Conclusion and Discussion

We have presented an active object library based on effect handlers and proved the correctness of its implementation principles. To prove this correctness, we expressed the implementation as a translation from a future calculus to an effect calculus and proved a bisimulation relation between the source and the transformed program. This illustrates that effects are a very general and versatile construct which can be leveraged to implement concurrency constructs as libraries, including futures. We discuss below a few alternatives that we considered and, more generally, extensions of this work we envision.

Deep and Shallow Handlers As highlighted at multiple points, we use *shallow* effect handlers, both in our implementation and in our formal development. Shallow effect handlers are not automatically reinstalled upon resuming a continuation, while deep handlers are automatically reinstalled.

In theory, Hillerström and Lindley [12] show that both deep and shallow handlers are equivalent, and showcase code transformation from one to the other. In addition, OCaml provides both versions. In practice, however, for the purpose of implementing a scheduler, shallow handlers offer numerous advantages. First, they make recursion in the *continue* function uniform over all tasks, be they continuations or new tasks. Furthermore, since they allow precise control over when handlers are installed, we can ensure that we never install nested handlers. In our implementation, this was essential to make *continue* and *run* tail-recursive.

Unfortunately, shallow handlers are a bit more delicate to implement for language designers. Furthermore, deep handlers admit a more precise small-step semantics [19]. It remains to be seen if the deep version of our scheduler can be expressed as elegantly as the one showcased in our formalisation.

Relation to Existing Promise-as-effect Libraries To develop our active object library, we made our own implementation of promises. This was convenient, as full-control allowed us to tie both together, which was essential for implementing *forward*, notably.

However, implementing an industrial-strength promise library with efficient scheduling, parallelism, and system integration is a significant task. Making several such libraries work together is delicate. In practice, `eio` [15] is trending towards being the standard promise library in OCaml.

Now that we formalised our semantics independently, one of the next steps is to adapt our developments to rely on an existing scheduling library. There are two difficulties here:

– Adapting to different underlying primitives (`eio` uses "suspend", similar to a form of yielding, and "fork" to create new promises).
– Finding a way to extend the scheduler implemented by an existing library, accessing its internal state, without completely breaking its invariants, nor breaking abstraction.

Optimisation on Forward As we mentioned in Section 3, forward is a construct that allows efficient delegation of asynchronous method invocations by making shortcuts when a future is resolved with another one [6]. For simplicity, we decided not to specify forward in our formal development. Its specification and proof is rather straightforward, by introducing an additional effect. In the future, in addition to this formal aspect, we would like to experiment with introducing delegated method calls automatically, following the analogy with tail-call optimisations.

Hiding Semantics and Middle-step Reductions Proof of correctness of translations between languages and calculi often reduce to simulation or bisimulation proofs [6, 5, 16] between a source program and a transformed program. Often, it is however necessary for the transformed program to do more steps than the original one. These additional internal steps are necessary to maintain internal information on the program state. Sometimes, even the source program must also do some internal steps. The usual tool to prove the equivalence in this case is to use a weak bisimulation that "ignores" some steps marked as internal. However, weak bisimulations do not guarantee the preservation of all program properties, in particular liveness properties [8]. In such situations, some previous work prove branching bisimilarity which is stronger but not always sufficient.

In this article, we developed a new "hiding" semantics and a middle-step reduction which executes one non-hidden step, followed by as many hidden steps as possible. This allows us to decide exactly in the specification of the translation which code is "administrative" and which code must really be synchronised. Naturally, in our context, such code is deterministic.

While we developed this in an ad-hoc manner here, we believe this approach can be adapted to many other program translations, simplifying simplifying the proof of correctness for compilers, and program transformations in general.

References

1. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: Proc. Symp. on Artificial Intelligence and Programming Languages, pp. 55–59. New York, NY, USA (1977)
2. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebraic Methods Program. 84(1), 108–123 (2015), https://doi.org/10.1016/j.jlamp.2014.02.001
3. de Boer, F., Din, C.C., Fernandez-Reyes, K., Hähnle, R., Henrio, L., Johnsen, E.B., Khamespanah, E., Rochas, J., Serbanescu, V., Sirjani, M., Yang, A.M.:

A survey of active object languages. ACM Computing Surveys 50(5), 76:1–76:39 (Oct 2017), article 76
4. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 123–134. ACM Press (2004)
5. Chappe, N., Henrio, L., Maillé, A., Moy, M., Renaud, H.: An optimised flow for futures: From theory to practice. CoRR abs/2107.07298 (2021), https://arxiv.org/abs/2107.07298
6. Fernandez-Reyes, K., Clarke, D., Castegren, E., Vo, H.P.: Forward to a promising future. In: Di Marzo Serugendo, G., Loreti, M. (eds.) Coordination Models and Languages. pp. 162–180. Springer International Publishing, Cham (2018)
7. Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E.B., Wrigstad, T.: Godot: All the Benefits of Implicit and Explicit Futures. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 2:1–2:28. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019), http://drops.dagstuhl.de/opus/volltexte/2019/10794, distinguished artefact
8. Graf, S., Sifakis, J.: Readiness semantics for regular processes with silent actions. In: Ottmann, T. (ed.) Automata, Languages and Programming. pp. 115–125. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
9. Halstead, Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(4), 501–538 (1985)
10. Henrio, L.: Data-flow Explicit Futures. Research report, I3S, Université Côte d'Azur (Apr 2018), https://hal.archives-ouvertes.fr/hal-01758734
11. Henrio, L., Johnsen, E.B., Pun, V.K.I.: Active objects with deterministic behaviour. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12546, pp. 181–198. Springer (2020)
12. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Ryu, S. (ed.) Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11275, pp. 415–435. Springer (2018), https://doi.org/10.1007/978-3-030-02768-1_22
13. Johnsen, E.B., Blanchette, J.C., Kyas, M., Owe, O.: Intra-object versus inter-object: Concurrency and reasoning in Creol. In: Proc. 2nd Intl. Workshop on Harnessing Theories for Tool Support in Software (TTSS'08). Electronic Notes in Theoretical Computer Science, vol. 243, pp. 89–103. Elsevier (Jul 2009)
14. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal

Methods for Components and Objects (FMCO 2010). LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
15. Leonard, T., Ferris, P., Haesbaert, C., Pluvinage, L., Karvonen, V., Parimala, S., Sivaramakrishnan, K., Balat, V., Madhavapeddy, A.: Eio 1.0 – effects-based io for ocaml 5. OCaml Workshop (2023)
16. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages. pp. 42–54. ACM Press (2006), http://xavierleroy.org/publi/compiler-certif.pdf
17. Liskov, B., Shrira, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. p. 260–267. PLDI '88, Association for Computing Machinery, New York, NY, USA (1988), https://doi.org/10.1145/53990.54016
18. Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theoretical Computer Science 364(3), 338–356 (2006)
19. Sieczkowski, F., Pyzik, M., Biernacki, D.: A general fine-grained reduction theory for effect handlers. Proc. ACM Program. Lang. 7(ICFP) (aug 2023), https://doi.org/10.1145/3607848
20. Sivaramakrishnan, K.C.: https://github.com/kayceesrk/ocaml5-tutorial (????), accessed: 2023-05-30
21. Sivaramakrishnan, K.C., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. Proc. ACM Program. Lang. 4(ICFP), 113:1–113:30 (2020), https://doi.org/10.1145/3408995
22. Sivaramakrishnan, K.C., Dolan, S., White, L., Kelly, T., Jaffer, S., Madhavapeddy, A.: Retrofitting effect handlers onto ocaml. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 206–221. ACM (2021), https://doi.org/10.1145/3453483.3454039
23. Taura, K., Matsuoka, S., Yonezawa, A.: Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In: Proceedings of the DIMACS workshop on Specification of Parallel Algorithms. pp. 275–292. American Mathematical Society (1994)
24. Vouillon, J.: Lwt: a cooperative thread library. In: Sumii, E. (ed.) Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008. pp. 3–12. ACM (2008), https://doi.org/10.1145/1411304.1411307

A Proof of the bisimulation theorem (Theorem 2)

A proof of bisimulation involves two simulation proofs for the same relation. We detail the proof for the first direction: the behaviour of the compiled program is one of the behaviours of the original one. This direction is more complex because

of the middle-step semantics and is also more important as it states that the behaviour of the compiled program is a valid one. The other direction is done very similarly with the same arguments as the ones used in the first direction. It however has a different structure as the SOS semantics provides more different cases (but the proof below often needs to distinguish cases according to the current state of the configuration, leading to a similar set of cases overall). We omit the other direction.

Consider $\sigma_1, P_1 \ \mathrm{R}_{\parallel} \ \sigma_1', F_1, P_1'$, and $\sigma_1, P_1 \overset{\Longrightarrow}{\scriptstyle ||} \sigma_2, P_2$. Let i be the thread identifier of the thread involved in the reduction $\overset{\Longrightarrow}{\scriptstyle ||}$ (in case of spawn i is the thread that performs the spawn).

We have $P_1 = Q_1 \parallel e^i$ and $P_1' = Q_1' \parallel s^i$ for some Q_1 and Q_1'. Additionally, $\sigma_1, Q_1 \ \mathrm{R}_{\parallel} \ \sigma_1', F_1, Q_1'$ and $\sigma_1, e \ \mathrm{R}_i \ \sigma_1', F_1, s$.

We do a case analysis on the rule used to prove the R_i relation; two cases are possible:

Continue:

$$\textsc{Continue}\ \frac{\sigma_1, \ell_{threads} \ \mathrm{R}_e \ \sigma_1', F_1}{\sigma_1, C_c[\ell_{threads}, f, [\![e']\!]_e, i] \ \mathrm{R}_i \ \sigma_1', F_1, (f \mapsto e')}$$

In this case, the top level of continue is a *handle* thanks to the context C_c. $\sigma_1, P_1 \overset{\Longrightarrow}{\scriptstyle ||} \sigma_2, P_2$ can result from three possible rules (modulo a SEQ rule at the configuration level and a λ-calculus context rule to reach the reducible statement):

HANDLE-RETURN $[\![e']\!]_e$ must be of the form v (and is inside a handle because of C_c).

We have $\sigma_1, P_1 \overset{\Longrightarrow}{\scriptstyle ||} \sigma_2, P_2$. Its first visible reduction rule must be:

$$\dfrac{\dfrac{\dfrac{(x \mapsto e_2) \in h}{\sigma_1, \texttt{handle}(v)\{h\} \longrightarrow \sigma_1, e_2\,[x \leftarrow v]}\ \textsc{handle-return}}{\sigma_1, C_{rec}\,[\texttt{handle}(v)\{h\}] \longrightarrow \sigma_1, C_{rec}\,[e_2\,[x \leftarrow v]]}\ \textsc{context}}{\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_1, Q_1 \parallel C_{rec}\,[e_3]^i}\ \textsc{seq}$$

Where:

C_{rec} the "let ... rec" context

h the effect handlers defined in Continue

$$\begin{aligned} e &= C_c[\ell_{threads}, f, [\![e']\!]_e, i] \\ &= C_{rec}[\texttt{handle}(v)\{h\}] \end{aligned}$$

$$e_3 = \begin{array}{l} \ell_{threads}[f] \leftarrow \mathbb{V}(v); \\ \mathit{run}(i) \end{array}$$

The hidden rules then update the appropriate task in the store and start the *run* function. Overall, we obtain:

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, i] \Longrightarrow_{||} \sigma_2, Q_1 \parallel C_r[\ell_{threads}, i]^i$$

Where

$$\sigma_2 = \sigma_1\left[\ell_{threads} \mapsto \sigma_1(\ell_{threads})\left[f \mapsto \mathbb{V}(v)\right]\right]$$

Since $e = C_{rec}[\texttt{handle}(v)\{h\}]$, by case analysis on the compilation rules, we must have the source expression $e' = v'$ be a FUT value with $v = [\![v']\!]_e$. Then we have:

$$\frac{\dfrac{}{\sigma_1', F_1, (f \mapsto v')^i \longrightarrow \sigma_1', F_1[f \mapsto v'], \texttt{Idle}^i}\ \text{RETURN}}{\sigma_1', F_1, Q_1' \parallel (f \mapsto v')^i \longrightarrow_{||} \sigma_1', F_1[f \mapsto v'], Q_1' \parallel \texttt{Idle}^i}\ \text{ONE-STEP}$$

We then need to establish that the new future map and stores are in relation, i.e., $\sigma_2, \ell_{threads}\ \mathrm{R}_e\ \sigma_1', F_1[f \mapsto v']$.
We recall the ENV rule below:

ENV

$$\frac{\begin{array}{c} F_e = F_{e,1} \uplus F_{e,2} \\ \forall f \in \mathrm{Dom}(F_v).\ F_v(f) \text{ is a value} \\ \forall f \in \mathrm{Dom}(F_e).\ F_e(f) \text{ is not a value} \\ T_{e,1} = [f' \mapsto \mathbb{C}(\lambda().\,[\![e]\!]_e) \mid F_{e,1}(f') = e] \\ T_{e,2} = \left[f' \mapsto \mathbb{C}\left(\lambda().((\lambda x.C[x])\ e)\right) \mid [\![F_{e,2}(f')]\!]_e = C[e]\right] \\ T_v = [f' \mapsto \mathbb{V}([\![v]\!]_e) \mid F_v(f') = v] \end{array}}{\sigma_{base} \cup \{\ell_{threads} \mapsto T_{e,1} \uplus T_{e,2} \uplus T_v\}, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_{base}, F_e \uplus F_v}$$

By inversion on $\sigma_1, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_1', F_1$, we obtain three maps $T_{e,1} \uplus T_{e,2} \uplus T_v$ that ensure the relation. We extend T_v so that $T_v[f] \mapsto \mathbb{V}(v)$ to obtain the relation.
Recall that $v = [\![v']\!]_e$; this is sufficient to conclude that

$$\sigma_2, Q_1 \parallel C_r[\ell_{threads}, i]^i \qquad \mathrm{R}_{\parallel} \qquad \sigma_1', F_1[f \mapsto v'], Q_1' \parallel \texttt{Idle}^i$$

HANDLE-STEP $[\![e']\!]_e$ must be of the form e_1 where e_1 can only be reduced by a λ-calculus reduction.
We have $\sigma_1, P_1 \Longrightarrow_{||} \sigma_2, P_2$. Its first visible reduction rule must be:

$$\frac{\dfrac{\dfrac{\sigma_1, e_1 \longrightarrow \sigma_2, e_2}{\sigma_1, e_1 \longrightarrow \sigma_2, e_2}\ \text{HANDLE-STEP}}{\sigma_1, C_{rec}\left[\texttt{handle}(e_1)\{h\}\right] \longrightarrow \sigma_2, C_{rec}\left[\texttt{handle}(e_2)\{h\}\right]}\ \text{CONTEXT}}{\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_2, Q_1 \parallel C_{rec}\left[e_3\right]^i}\ \text{SEQ}$$

Where:

$$\begin{aligned} &C_{rec} \text{ the "let ... rec" context}\\ &\quad h \text{ the effect handlers defined in Continue}\\ &\quad e = C_c[\ell_{threads}, f, [\![e']\!]_e, i] = C_{rec}[\texttt{handle}(e_1)\{h\}]\\ &\quad e_3 = \texttt{handle}(e_2)\{h\} \end{aligned}$$

The translation leave λ-calculus terms unchanged, without any hiding, thus there are no follow up hidden rules.
Overall, we obtain:

$$\frac{\sigma_1, [\![e']\!]_e \longrightarrow \sigma_2, e_2}{\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, i] \underset{||}{\Longrightarrow} \sigma_2, Q_1 \parallel C_c[\ell_{threads}, f, e_2, i]}$$

We know that $\sigma_1, \ell_{threads}\ \mathrm{R}_e\ \sigma_1', F_1$. By definition, this means that $\sigma_1 = \sigma_1' \cup \{\ell_{threads} \mapsto T\}$ for some map T. By definition of the translation, $\ell_{threads}$ is not accessible by user code, and thus left unchanged by the reduction on $[\![e']\!]_e$. As such, we have:

$$\sigma_2 = \sigma_2' \cup \{\ell_{threads} \mapsto T\} \qquad \sigma_1', [\![e']\!]_e \longrightarrow \sigma_2', e_2$$

By case analysis on the translation and the λ-calculus reduction rules, e' must be reduced by the same λ-calculus reduction rule than $[\![e']\!]_e$. Thus:

$$\frac{\dfrac{\sigma_1', e' \longrightarrow \sigma_2', e_2'}{\sigma_1', F_1, (f \mapsto e')^i \longrightarrow \sigma_2', F_1, (f \mapsto e_2')^i}\ \textsc{step}}{\sigma_1', F_1, Q_1' \parallel (f \mapsto e')^i \longrightarrow_{||} \sigma_2', F_1, Q_1' \parallel (f \mapsto e_2')^i}\ \textsc{one-step}$$

This case analysis and by determinism of our λ-calculus, we have $[\![e_2']\!]_e = e_2$. We also have $\sigma_2, \ell_{threads}\ \mathrm{R}_e\ \sigma_2', F_1$.
This is sufficient to conclude that

$$\sigma_2, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, i]^i\ \ \mathrm{R}_{\parallel}\ \ \sigma_2', F_1, Q_1' \parallel C_c[\ell_{threads}, f, [\![e_2']\!]_e, i]$$

HANDLE-EFFECT $[\![e']\!]_e$ must be of the form $C[\texttt{throw}(E(x))$ (and is inside a handle because of C_c). We distinguish by the effect captured:

***Async*(*job*, t')** We have $\sigma_1, P_1 \underset{||}{\Longrightarrow} \sigma_2, P_2$. Its first visible reduction rule must be:

$$\frac{\begin{array}{c}\textsc{seq+handle-effect+context}\\ (Async(job, t'), k' \mapsto e_2) \in h \qquad Async \notin \text{captured\_effects}(C)\end{array}}{\begin{array}{c}\sigma_1, C_{rec}\left[\texttt{handle}(C[\texttt{throw}(Async(\ \lambda().\ e'', t))])\{h\}\right]\\ \longrightarrow \sigma_1, C_{rec}\left[e_2\,[t' \leftarrow t]\left[job \leftarrow\ \lambda().\ e''\right][k' \leftarrow \lambda y.C[y]]\right]\end{array}}$$
$$\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_1, Q_1 \parallel C_{rec}\,[e_3]^i$$

Where:

$$
\begin{aligned}
&C_{rec} \text{ the “let ... rec” context}\\
&\quad h \text{ the effect handlers defined in Continue}\\
&\quad e = C_c[\ell_{threads}, f, [\![e']\!]_e, i]\\
&\qquad = C_{rec}[\texttt{handle}(C[\texttt{throw}(Async(\ \lambda().\ e'', t))])\{h\}]\\
&e_3 = \ \texttt{let}\ fut' = (t, fresh())\ \texttt{in}\\
&\qquad \ell_{threads}[fut'] \leftarrow \mathbb{C}(\ \lambda().\ e'');\\
&\qquad continue(f, \lambda().(\lambda y.C[y])(fut'), i)
\end{aligned}
$$

By definition of the translation, and because the reduction is possible, the arguments of the `Async` effect must be a thunk task, and its second argument must be a thread identifier (it can be an expression but this one is entirely evaluated before triggering the effect). This as some consequences on the considered FUT configuration, e.g. e' is of the form $\texttt{AsyncAt}(e_0, t)$. Additionally, t is the same on both side as thread identifiers are preserved by the translation (this can be proven by case analysis on the definition of R_i).
The hidden rules apply then update the suspended tasks in the store and start the *continue* function. The last hidden reduction rule is the beta-reduction that de-thunks the continuation $\lambda().(\lambda y.C[y])(fut')$ inside the handler of continue and puts fut' back into the invocation context.
Overall, we obtain:

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, i] \Longrightarrow_{||} \sigma_2, Q_1 \parallel C_c[\ell_{threads}, f, C[fut'], i]^i$$

Where

$$\sigma_2 \quad = \quad \sigma_1\Big[\ell_{threads} \quad \mapsto \quad \sigma_1(\ell_{threads})\ \Big[fut' \mapsto \mathbb{C}(\ \lambda().\ e'')\Big]\ \Big]$$

Since $e = C_{rec}[\texttt{handle}(C[\texttt{throw}(Async(\ \lambda().\ e'', t))])\{h\}]$, by case analysis on the compilation rules, we must have the source expression $e' = C_1[\texttt{asyncAt}(e'_1, t)]$ where $C = [\![C_1]\!]_e$ and $e'' = [\![e'_1]\!]_e$ by Lemma 1. Note also that the set of future identifiers are the same in the FUT program and in its translation, and thus $fut' = (t, fresh())$ is a fresh future in the FUT configuration. Then we have:

$$\frac{fut' = (t, lf) \qquad fut' \notin \mathrm{Dom}(F_1)}{\sigma'_1, F_1, Q'_1 \parallel (f \mapsto C_1[\texttt{asyncAt}(e'_1, t)])^i \longrightarrow_{||} \sigma'_1, F_1\left[fut' \mapsto e'_1\right], Q'_1 \parallel (f \mapsto C_1[fut'])^i}\ \text{ASYNC+ONE-STEP}$$

We then need to establish that the new future map and stores are in relation, i.e., $\sigma_2, \ell_{threads}\ \mathrm{R}_e\ \sigma_1', F_1\left[fut' \mapsto e_1'\right]$.
We recall the ENV rule below:

ENV

$$\frac{\begin{array}{c} F_e = F_{e,1} \uplus F_{e,2} \\ \forall f \in \mathrm{Dom}(F_v).\ F_v(f) \text{ is a value} \\ \forall f \in \mathrm{Dom}(F_e).\ F_e(f) \text{ is not a value} \\ T_{e,1} = [f' \mapsto \mathbb{C}(\lambda().\ [\![e]\!]_e) \mid F_{e,1}(f') = e] \\ T_{e,2} = \left[f' \mapsto \mathbb{C}\left(\lambda().((\lambda x.C[x])\ e)\right) \mid [\![F_{e,2}(f')]\!]_e = C[e]\right] \\ T_v = [f' \mapsto \mathbb{V}([\![v]\!]_e) \mid F_v(f') = v] \end{array}}{\sigma_{base} \cup \{\ell_{threads} \mapsto T_{e,1} \uplus T_{e,2} \uplus T_v\}, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_{base}, F_e \uplus F_v}$$

By inversion on $\sigma_1, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_1', F_1$, we obtain tree maps $T_{e,1} \uplus T_{e,2} \uplus T_v$ that ensure the relation. We then extend $T_{e,1}$ so that $\ell_{threads}[fut'] \mapsto \mathbb{C}(\lambda().\ [\![e_1']\!]_e)$ to obtain the relation.
This is sufficient to conclude that

$$\sigma_2, Q_1 \parallel C_c[\ell_{threads}, f, C[fut'], i]^i \quad \mathrm{R}_{\parallel} \quad \sigma_1', F_1\left[fut' \mapsto e_1'\right], Q_1' \parallel (f \mapsto C_1[fut'])^i$$

Get(f') We have $\sigma_1, P_1 \Longrightarrow_{||} \sigma_2, P_2$. Its first visible reduction rule must be:

SEQ+HANDLE-EFFECT+CONTEXT

$$\frac{\begin{array}{c} (Get(fut_g), k' \mapsto e_2) \in h \qquad Get \notin \text{captured\_effects}(C) \\ \hline \sigma_1, C_{rec}\left[\texttt{handle}(C[\texttt{throw}(Get(f'))])\{h\}\right] \\ \longrightarrow \sigma_1, C_{rec}\left[e_2\left[fut_g \leftarrow f'\right]\left[k' \leftarrow \lambda y.C[y]\right]\right] \end{array}}{\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_1, Q_1 \parallel C_{rec}\left[e_3\right]^i}$$

Where[8]:

C_{rec} the "let ... rec" context

h the effect handlers defined in Continue

$$\begin{aligned} e &= C_c[\ell_{threads}, f, [\![e']\!]_e, i] \\ &= C_{rec}[\texttt{handle}(C[\texttt{throw}(Get(f'))])\{h\}] \end{aligned}$$

$$e_3 = \begin{array}{l} \texttt{let } v = poll(f') \texttt{ in} \\ continue(f, \lambda().((\lambda y.C[y])\ v), i) \end{array}$$

The argument of the `Get` effect must be a future reference that is totally evaluated for the rule to succeed. If it is not a future the evaluation of *poll* fails. If it is not fully evaluated, the reduction should first occur inside the argument of the `Get` effect.

[8] a few substitutions have occurred inside *poll* by definition of C_c. We omit them here not to clutter the proof.

Details on poll reductions At this point, we look at hidden reductions, which must start in the body of *poll*. If the future is unresolved, *poll* loops forever and the medium step reduction diverges. This means either that the future never resolves, and this divergence in Eff simulates a deadlock in Fut; or that we could make reductions in other threads to resolve the deadlock. In the second case, the semantics for Eff would interleave loops in *poll* and reduction in other threads. Such interleaving is equivalent to triggering the `Get` event at the end, with a single loop in *poll*. The current theorem only consider this last interleaving. Overall, if there is a medium step reduction it means that the future is resolved.

In this case, the future has been resolved, and, by bisimilarity on the stores (R_e) we have $F_1(f') = v$ and $\sigma_1(\ell_{threads})[f'] = v$ for some v. We obtain after a couple of steps of beta-reduction:

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, i] \Longrightarrow_{||} \sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[v], i]^i$$

Since $e = C_{rec}[\texttt{handle}(C[\texttt{throw}(\mathit{Get}(f'))])\{h\}]$, by case analysis on the compilation rules, we have $e' = C_1[\texttt{get}(f')]$ where $C = [\![C_1]\!]_e$ by Lemma 1. Then we have:

GET+ONE-STEP

$$\frac{(f' \mapsto v) \in F_1}{\sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[\texttt{get}(f')])^i \longrightarrow_{||} \sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[v])^i}$$

This is sufficient to conclude that

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[v], i]^i \quad \mathrm{R}_{\parallel} \quad \sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[v])^i$$

Await(f') The case when the awaited future is resolved is similar to the case of the `Get` effect just above. We only detail the proof in case the future is still unresolved.
We have $\sigma_1, P_1 \Longrightarrow_{||} \sigma_2, P_2$. Its first visible reduction rule must be:

SEQ+HANDLE-EFFECT+CONTEXT

$$\frac{\begin{array}{c}(\mathit{Await}(\mathit{fut}_a), k' \mapsto e_2) \in h \qquad \mathit{Await} \notin \mathrm{captured\_effects}(C) \\ \hline \sigma_1, C_{rec}\,[\texttt{handle}(C[\texttt{throw}(\mathit{Await}(f'))])\{h\}] \\ \longrightarrow \sigma_1, C_{rec}\,[e_2\,[\mathit{fut}_a \leftarrow f']\,[k' \leftarrow \lambda y.C[y]]]\end{array}}{\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_1, Q_1 \parallel C_{rec}\,[e_3]^i}$$

Where:

C_{rec} the "let ... rec" context

h the effect handlers defined in Continue

$$\begin{aligned} e &= C_c[\ell_{threads}, f, [\![e']\!]_e, i] \\ &= C_{rec}[\texttt{handle}(C[\texttt{throw}(Await(f'))])\{h\}] \end{aligned}$$

$$\begin{aligned} e_3 = \; & \texttt{match}\ \ell_{threads}[f']\{ \\ & \mid \mathbb{V}(v) \mapsto continue(f, \lambda().((\lambda y.C[y])\ v)) \\ & \mid \_ \mapsto\ \texttt{let}\ k''() = (\lambda y.C[y])\ (\texttt{throw}(Await(f')))\ \texttt{in} \\ & \qquad \ell_{threads}[f] \leftarrow \mathbb{C}(k''); run(i) \\ & \} \end{aligned}$$

Like in the `Get` case, the argument of the `Await` effect must be a future reference that is totally evaluated for the rule to succeed. When the future is unresolved, $\ell_{threads}[f']$ is not a value (it is not mapped or mapped to a $\mathbb{C}$). By definition of R_e we necessarily have: $\nexists v.\,(f' \mapsto v) \in F_1$. Then a few hidden beta reduction steps lead to the following configuration:

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, [\![e']\!]_e, t] \Longrightarrow_{||} \sigma_2, Q_1 \parallel C_r[\ell_{threads}, i]^i$$

Where

$$\begin{aligned} \sigma_2 = \sigma_1\Big[\ell_{threads} \mapsto \sigma_1(\ell_{threads}) \\ \big[f \mapsto \mathbb{C}\big(\lambda().((\lambda y.C[y])\ (\texttt{throw}(Await(f'))))\big)\big]\Big] \end{aligned}$$

Since $e = C_{rec}[\texttt{handle}(C[\texttt{throw}(Await(f'))])\{h\}]$, by case analysis on the compilation rules, we have $e' = C_1[\texttt{await}(f')]$ where $C = [\![C_1]\!]_e$ by Lemma 1. Thus on the FUT side, we have:

$$\frac{\text{AWAIT-YIELD+ONE-STEP}\quad \nexists v.\,(f' \mapsto v) \in F_1}{\begin{array}{c}\sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[\texttt{await}(f')])^i \\ \longrightarrow_{||} \sigma_1', F_1\,[f \mapsto C_1[\texttt{await}(f')]]\,, Q_1' \parallel \texttt{Idle}^i\end{array}}$$

We easily obtain that $\sigma_2, \ell_{threads}\ \ \mathrm{R}_e\ \ \sigma_1', F_1\,[f \mapsto C_1[\texttt{await}(f')]]$ by expanding the environment $T_{e,2}$ in the ENV rule.
With the arguments above and the case RUN of $\mathrm{R}_{\parallel}$ we conclude:

$$\sigma_2, Q_1 \parallel C_r[\ell_{threads}, i]^i \quad \mathrm{R}_{\parallel} \quad \sigma_1', F_1\,[f \mapsto C_1[\texttt{await}(f')]]\,, Q_1' \parallel \texttt{Idle}^i$$

Spawn() We have $\sigma_1, P_1 \Longrightarrow_{||} \sigma_2, P_2$. Its first visible reduction rule must be:

$$\textsc{seq+handle-effect+context}\quad \frac{(\mathit{Spawn}(), k' \mapsto e_2) \in h \qquad \mathit{Spawn} \notin \text{captured\_effects}(C)}{\sigma_1, C_{rec}[\texttt{handle}(C[\texttt{throw}(\mathit{Spawn}())])\{h\}] \longrightarrow \sigma_1, e_2\,[k' \leftarrow \lambda y.C[y]]}}{\sigma_1, Q_1 \parallel e^i \longrightarrow_{||} \sigma_1, Q_1 \parallel e_2^i}$$

With: C_{rec} the "let ... rec" context of the continue handler inside C_c, h the effect handlers defined in Continue, additionally:

$$\begin{aligned} e &= C_{rec}[\texttt{handle}(C[\texttt{throw}(\mathit{Spawn}())])\{h\}] \\ &= C_c[\ell_{threads}, f, [\![e']\!]_e, t] \\ e_2 &= \texttt{let } t' = \texttt{spawn}(\mathit{run}) \texttt{ in } \mathit{continue}(\mathit{fut}, \lambda().k'(t'), t) \end{aligned}$$

The first hidden rule applied is

$$\textsc{spawn (hidden)}\quad \frac{\mathit{tid} \notin \mathit{tids}(P) \cup \{i\}}{\sigma_1, Q_1 \parallel C_2[\texttt{spawn}(\mathit{run})]^i \longrightarrow_{||} \sigma_1, Q_1 \parallel C_2[\mathit{tid}]^i \parallel C_c[(\mathit{run}\ \mathit{tid})]^{\mathit{tid}}}$$

Where $e_2 = C_2[\texttt{spawn}(\mathit{run})]$. This is followed by steps of beta reduction to reduce the $\texttt{let } t' = \ldots$ construct, trigger continue, pass the associated tid and de-thunk the $\lambda().\lambda y.C[y](\mathit{tid})$ inside continue. We obtain the following configuration

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[\mathit{tid}], t]^i \parallel C_c[(\mathit{run}\ \mathit{tid})]^{\mathit{tid}}$$

Finally, by a step of beta reduction in the thread tid we obtain the right evaluation context C_r

$$\sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[\mathit{tid}], t]^i \parallel C_r[\ell_{threads}, \mathit{tid}]^{\mathit{tid}}$$

This configuration is not reducible by a hidden transition. Thus

$$\begin{aligned} &\sigma_1, C_{rec}[\texttt{handle}(C[\texttt{throw}(\mathit{Spawn}())])\{h\}] \\ &\quad \Longrightarrow_{||} \sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[\mathit{tid}], t]^i \parallel C_r[\ell_{threads}, \mathit{tid}]^{\mathit{tid}} \end{aligned}$$

By case analysis on the terms involved in $\sigma_1, P_1 \ \mathrm{R}_{\parallel}\ \sigma_1', F_1, P_1'$ we have $e' = C_1[\texttt{spawn}()]$ where $C = [\![C_1]\!]_e$ by Lemma 1. We then have:

$$\textsc{spawn}\quad \frac{\mathit{tid} \notin \mathit{tids}(Q_1') \cup \{i\}}{\sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[\texttt{spawn}()])^i \longrightarrow_{||} \sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[\mathit{tid}])^i \parallel \texttt{Idle}^{\mathit{tid}}}$$

Note that by definition of $\mathrm{R}_{\parallel}$ the set of used thread identifiers is the same in both configurations, wo we can take the same fresh *tid*. Note also that the store and the future map are unchanged. Comparing thread by thread, we can directly apply rule RUN and rule CONT for the two processes *tid* and i, which leads to the conclusion:

$$\begin{array}{r} \sigma_1, Q_1 \parallel C_c[\ell_{threads}, f, C[tid], t]^i \parallel C_r[\ell_{threads}, tid]^{tid} \\ \mathrm{R}_{\parallel} \quad \sigma_1', F_1, Q_1' \parallel (f \mapsto C_1[tid])^i \parallel \texttt{Idle}^{tid} \end{array}$$

Run:

$$\textsc{run}\ \frac{\sigma_1, \ell_{threads} \ \ \mathrm{R}_e \ \ \sigma_1', F_1}{\sigma_1, C_r[\ell_{threads}, i] \ \ \mathrm{R}_i \ \ \sigma_1', F_1, \texttt{Idle}}$$

The only first applicable rule is the pop operation reduction that picks a new available thread:

$$\begin{array}{r} \sigma, C_r[\ell_{threads}, i] \xrightarrow{\mathrm{pop}}_{\mathrm{h}} Run\,[t \leftarrow i] \\ \xrightarrow{\tau}_{\mathrm{h}}{}^{*} \sigma_2, C_c[\ell_{threads}, f_2, e_2, i] \end{array}$$

Note that pop ensures that f_2 is of the form $f_2 = (i, lf)$. Using only reductions in the thread i and such that: $\sigma_1(\ell_{threads})[f_2] = \mathbb{C}(\lambda().\,[\![F_1(f_2)]\!]_e)$[9] by definition of R_i and $e_2 = [\![F_1(f_2)]\!]_e$[10] by definition of pop. Note that the last step of reduction is inside continue and de-thunks the new task $((\lambda().e_2()) \longrightarrow e_2)$[11]. We additionally have:

$$\sigma_2 = \sigma_1[\ell_{threads} \mapsto \sigma_1(\ell_{threads}) \setminus f_2]$$

From the points above, we obtain (with $f_2 = (i, lf)$):

$$\textsc{schedule}\ \frac{(f_2 \mapsto F_1(f_2)) \in F_1 \qquad F_1(f_2) \text{ is not a value}}{\sigma_1', F_1, Q_1' \parallel \texttt{Idle}^i \longrightarrow_{||} \sigma_1', F_1 \setminus f_2, Q_1' \parallel (f_2 \mapsto F_1(f_2))^i}$$

Note that $F_1(f_2)$ is not a value by construction of the equivalence on stores (Figure 16). Finally (the equivalence on the store can be trivially checked):

$$\textsc{Continue}\ \frac{\sigma_2, \ell_{threads} \ \ \mathrm{R}_e \ \ \sigma_1', F_1 \setminus f_2}{\sigma_2, C_c[\ell_{threads}, f_2, [\![F_1(f_2)]\!]_e, i] \ \ \mathrm{R}_i \ \ \sigma_2', F_1 \setminus f_2, (f_2 \mapsto F_1(f_2))}$$

This immediately concludes by adding the other threads (in Q_1 and Q_1') and obtaining the $\mathrm{R}_{\parallel}$ relation on the obtained configurations. □

[9] resp. $\sigma_1(\ell_{threads})[f_2] = \mathbb{C}\,(\lambda().((\lambda x.C[x])\ e))$
[10] resp. $[\![F_1(f_2)]\!]_e = C[e]$ and $e_2 = \mathbb{C}\,(\lambda().((\lambda x.C[x])\ e))$
[11] resp. with two steps of beta-reductions