Formalising Futures and Promises in Viper

Cinzia Giusto, Loïc Germerie Guizouarn, Ludovic Henrio, Etienne Lozes

Formalising Futures and Promises in Viper*

Cinzia Di Giusto[1], Loïc Germerie Guizouarn[1],
Ludovic Henrio[2], and Etienne Lozes[1]

[1] Université Côte d'Azur, Nice, France
{cinzia.di-giusto,loic.germerie-guizouarn,etienne.lozes}@univ-cotedazur.fr
[2] Université de Lyon, EnsL, UCBL, CNRS, Inria, LIP, Lyon, France
ludovic.henrio@cnrs.fr

Abstract

Futures and promises are respectively a read-only and a write-once pointer to a placeholder in memory. They are used to transfer information between threads in the context of asynchronous concurrent programming. Futures and promises simplify the implementation of synchronisation mechanisms between threads. Nonetheless they can be error prone as data races may arise when references are shared and transferred. We aim at providing a formal tool to detect those errors. Hence, in this paper we propose a proof of concept by implementing the future/promise mechanism in Viper: a verification infrastructure, that provides a way to reason about resource ownership in programs.

1 Introduction

Futures and promises are synchronisation primitives that are common in many programming languages. A future is a read-only placeholder for a value to be computed. It is usually filled with the result value of an asynchronous execution. A future can however be explicitly paired with a promise, i.e., a writable single assignment container which is used to set the value of the associated future.

Somehow like a communication channel that may carry only one message, a future/promise pair is used to transfer information between threads in the context of parallel executions. As an example, consider the Viper code of Listing 1. Technical details as well as necessary logical specification, represented in grey, can be left aside for now. Here, the same object (p) is used to act both as a future (read pointer) and a promise (write-once pointer). We will explain both the technical details and the way we represent the promise/future pair in the next section. Intuitively, a thread executing the method m is spawned and "returns" its result by resolving the promise p, while the main thread "awaits" this result with a blocking call to GET on future p. Later on, a second call to GET can be performed, it is non-blocking, and returns the same value as the first get. On the other side, multiple resolves on the same promise should be forbidden as a promise is a single-write entity. This helps to ensure determinacy of programs using futures. Aside from the issue of double resolves, the programmer also has to deal with standard pitfalls of concurrent programming like deadlocks and data races.

In this work we propose a Viper [24] library for future-manipulating programs that ensures standard safety properties, including memory safety, absence of races, and absence of double resolves. We introduce the idea of associating a "resource invariant" with each future/promise pair: the resolve primitive "inhales" the resource invariant, and the get primitive "exhales" it (in the simple case where there is just one get).

*This work has been supported by the French government, through the EUR DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE- 0004.

```
field value: Int

method m(p: Ref, x: Ref)
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 0
  requires acc(x.value)
{
  x.value := x.value * 2
  RESOLVE(p, x)
}

method main {
  var p: Ref
  var x: Ref
  var a: Ref
  var b: Ref
  x := new(value)
  p := new_promise(0)
  x.value := 4
  m(p, x)
  GET(p, a)
  GET(p, b)
}
```

Listing 1: Simple example with resolve, get, and asynchronous call

The promises we present in this paper are similar to the one implemented by `std::promise` in C++. The programmer can create a new promise, get the future associated to this promise, and set the value of the promise. Setting the value can happen only once, any later attempt will throw an error. In Java, the Future interface defines the consumer end of the concept, while the `CompletableFuture` implementation adds an explicit producer end placeholder. Method `get()` is a blocking read into the shared value, and `complete()` is the method allowing to set this value. An arbitrary number of calls to `complete()` are allowed, the method will return true for the first one and false for all subsequent ones, not modifying the first set value. The same two pointers approach is implemented in Rust: the crate `future` offers the function `promise` which creates a `Promise` and a `Complete`, where the `Promise` is the readonly placeholder and `Complete` is the write only pointer. This terminology differs from ours but the principle is the same: we get dissociated write and read pointer to a placeholder. As a last example of future/promise implementations that is worth mentioning is the JavaScript take. Promises work more as what we described as plain futures earlier, the difference being that instead of a `return` statement they use a `resolve` function which is a parameter of the spawn thread. The need for an explicit call to `resolve` makes our approach more suitable to reason about JavaScript programs than another one where a future would be completed implicitly at the end of a thread.

Our goal is to propose a general approach to promises and future so that all above concepts may be formalised and checked against properties. This paper represents a first step towards this objective. We propose a proof of concept by using the features of the verification framework Viper. The implementation of the library as well as the examples we discuss in this paper are available at [1].

Outline. We begin with a short introduction to Viper in Section 2. In Section 3 we present the promise/future mechanism, the challenges of its verification, and its implementation in Viper. In Section 4 we showcase our implementation on a classical producer-consumer example and a binary tree merging algorithm, both from Blelloch and Reid-Miller [4]. Finally, in Section 5 we discuss some limitations and future works.

Related works. Although not explicitly described in this paper, our final goal is to ensure properties of the promise/future mechanism via a suitable separation logic: as a matter of fact rules of the logic are represented here by proper pre and post conditions in the tool we have used. Hence the closest group of related works concerns the treatment of synchronisation mechanism in separation logic. Indeed, there has been a lot of work about axiomatising various specific synchronisation primitives, including locks [12, 14], channels [32, 21, 22, 15], or barriers [16, 17], to only quote a few. Parkinson pointed out that: "*there is a disturbing trend for each new library or concurrency primitive to require a new separation logic*" [26], and advocated that "*adding new concurrency libraries should simply be a matter of verification, not of new logics or metatheory*". Research therefore also focused on extending the expressive power of separation logic with features like permissions [5], rely-guarantee reasoning [31], higher-order [25, 27, 30], or views [8], only quoting a few of these extensions. These ideas are now well integrated in separation logic frameworks like Iris [19], Verifast [18] or Gillian [29]. Permission reasoning can even be encoded in tool that does not support this feature natively, like Why3 [10]. An example of such an encoding can be found in [6].

For what concerns, more specifically, futures and promises, a Hoare logic for reasoning about shared futures has been defined by Din and Owe [7]; the logic bases on "communication histories" of the objects but does not deal with resource ownership in a way comparable with separation logic. More recently, Gardner *et al.* addressed the verification of JavaScript programs that manipulate promises [28]; their approach builds on *transpiling* javascript code to the intermediate JSIL language, for which a symbolic execution engine has been developed. It seems rather technically involved for a newcomer to understand this transpiling process and guess how the original program should be specified to be correctly symbolically executed.

Finally, the promises and future mechanisms presents some similarities with prophecy variables [20], we leave for future work to proper study the connections between those two concepts.

2 Viper

Viper [24], standing for Verification Infrastructure for Permission-based Reasoning, is a verification infrastructure. More precisely, it offers an intermediate language and a verification condition generator. Programs written in Viper are checked against logical specifications, allowing to reason about ownership of resources.

A Viper program consists of a sequence of declaration of memory reference fields, predicates, functions and methods. Each memory reference is instantiated with all the declared fields. Permissions specify which fields and predicates may be accessed: `write` denotes full ownership, `none` no ownership and `wildcard` "some amount" of permission. Method declaration typically begins with a set of pre and post-conditions. Pre-conditions give the permissions as well as the properties (logical assertions) required to execute the body of the method, while post-conditions represent what the method call will ensure, in terms of permissions or properties, after its executions. The grey lines of Listing 1 are an illustration of those concepts. The keyword `acc` at line 6 is an assertion specifying total permission on `x.value`. Without this pre-condition, method `m` could not perform the update at line 8. The pre-condition at line 4 specifies full permission on a predicate: `promise(x)`.

Predicates allow to define composed assertion parametrised by some value (in `promise(x)`, for instance, `x`, the reference that is a promise). They describe some permissions, but in order to use those permissions they must be "unfolded". The `unfold` keyword transfers the ownership of resources from the predicate to the program state. The converse action, `fold`, asserts ownership of a predicate whose body is satisfied and consumes the resources captured by it (hence the program state loses all permissions on those resources). The last pre-condition, at line 5, uses this mechanism: it requires a specific field to be equal to 0, and to gain the right to access this field, a permission on this field contained in `promise(p)` is obtained by temporarily unfolding this predicate. Why this pre-condition is useful will be explained later.

To verify a program, Viper checks all methods independently. Permission on resources accessed in the body of a method must be granted by the pre-conditions of this method, and the program state at the end of the body must satisfy the post-conditions. A method call is only interpreted with respect to its logical specifications: it asserts and consumes the pre-conditions, and assumes the post-conditions. For example, in Listing 1, after the call to `m` in method `main`, because permission on `x.value` is required by `m` and not ensured as post-condition, this field cannot be accessed after line 20.

3 Verifying Programs Using Promises

In this section we will explain how promises and futures work, and how we can verify programs that use them. We will then present our implementation of these synchronisation primitives in Viper.

3.1 Promises and Futures

To encode the use of promise/future pairs, we need four ingredients:

- a way to start an asynchronous computation,
- a way to create the pair of pointers, we call `new_promise` the statement that performs this operation,
- a way to assign a value to the promise, we call this operation `resolve`,
- a way to read the value of a resolved future, we call this operation `get`.

The first two elements are self-explanatory. By definition, only one `resolve` is allowed for each promise. This prevents race-conditions between accesses to the promise. The `get` primitive is blocking until the future is assigned a value through its associated promise, then it returns this value.

A key consideration is that this communication mechanism is used to transfer information between asynchronous threads. This implies that in a heap modifying language, the transferred information could be a reference to memory. To verify programs using this mechanism, we have to be able to reason about how ownership is transferred alongside the information. We will focus on a case where the only way to synchronise and transfer information between asynchronous threads is the resolution of a promise followed by the `get` operation on its associated future. In the example discussed in the Introduction (Listing 1), to prevent data race we must ensure that `m` has exclusive access to `x.value`. But after the line `GET(p, a)`, we need to acknowledge

```
predicate promise(x: Ref){
  acc(x.locker, wildcard) &&
  acc(x.resolved, 1/2) &&
  acc(x.consumed, 1/2) &&
  acc(x.fut_value, 1/2) &&
  x.consumed == none &&
  x.resolved == false &&
  acc (x.inv_id, wildcard)
}

predicate future(x: Ref){
  acc(x.locker, wildcard) &&
  acc(x.inv_id, wildcard)
}

predicate resolved_future(x: Ref) {
  acc(x.locker, wildcard)  &&
  acc(x.resolved, wildcard) &&
  acc(x.fut_value, wildcard) &&
  x.resolved
}
```

Listing 2: Viper promise, future and resolved future predicates

the fact that `m` gave up the lock it had on this field, and allow subsequent instructions to access it.

To tackle this task, we introduce the concept of promise invariant, a logical specification of the information a future can contain. Each promise is associated with such an invariant. Resolving the promise asserts and consumes the invariant, while getting the future assumes it.

There are two caveats to take into consideration for this last operation: a future may be shared between threads, and a future may be read more than once in the same thread. To deal with the first case, we must ensure that the permission obtained on the invariant is equal to the permission owned on the future when calling `get`. As long as we can ensure no more than full ownership on a future is shared between every threads, we know that no more than full ownership on the invariant will be acquired.

If there are multiple calls to `get`, the same permission must not be obtained twice. This can be achieved by logically making the distinction between a future pointer that has not been read yet, and one that has been read at least once. We call a future pointer on which `get` was already called a "resolved future". The `get` operation consumes some permission on a future, and provides a similar amount of permission on a resolved future. We can call `get` on a resolved future as well, this operation does not consume the permission on this object, neither it provides permission on the promise invariant.

3.2 Implementation

The encoding of promises and futures is done via a single memory reference with 5 fields: `fut_value`, `resolved`, `consumed`, `inv_id` and `locker`. Once computed, the result is written in, and read from the field `fut_value`. Field `resolved` is a boolean flag set to true when the

```
predicate ip(v: Ref, prom: Ref){
  acc(prom.inv_id, wildcard) &&
  (prom.inv_id == 0 ? acc(v.value) :
   prom.inv_id == 1 ? (v == null ? true :
                                  acc(v.value) &&
                                  acc(v.next) &&
                                  future(v.next) &&
                                  unfolding future(v.next) in v.next.inv_id == 1) :
   prom.inv_id == 2 ? tree(v) :
   true)
}
```

Listing 3: Example of promise invariant implementation

promise is resolved. Field `consumed` is a logical variable used to remember how much ownership of the promise invariant has been consumed by the primitive get. The last two fields are used to implement the interplay between futures and promises, allowing the predicates we will define in the following paragraphs to ensure consistency of the reference through resolve and get calls.

More precisely, the role of promises and futures is encoded by three predicates. For a given reference `x`, `promise(x)` gives permission to treat `x` as a promise; `future(x)` gives permission to treat `x` as a future that has not been read yet, that is calling `get` on this future will assume the resource invariant; and `resolved_fut(x)` gives right to "get" a future without gaining permission on the promise invariant. These three predicates are shown in Listing 2.

To ensure consistency of the state of the reference representing the future, the permission to its fields are guarded by predicate `lock_inv`. It is a lock invariant, meaning that only one branch of a parallel composition may use the permission it provides. This predicate is shown in Listing 4, and it ensures that the thread owning it has a partial possession of the reference's fields. This allows a get call to access its fields to read them easily. When combined with the permission contained in the `promise(x)` predicate, the permissions provided by `lock_inv(x)` allow to modify fields `resolved` and `fut_value`. The disjunction at line 7 captures the different permissions that may be owned depending on whether the future is resolved or not. In the first case, this helps keeping track on the amount of permission still available for the promise invariant. The two macros `LOCK(x)` and `UNLOCK(x)` represent the locking (respectively unlocking) mechanism, providing or capturing the resources described by `lock_inv(x)`.

The last predicate we will mention here is `ip(val, fut)` (presented in Listing 3). It is the predicate that represents the permission transferred through a promise resolution. To be able to have a different invariant for different promises, the resources described by this predicate depend on the value of the field `inv_id`. This field is initialised when creating a new promise. The `ip` predicate must be defined for all the different promise invariant needed in a program. The implementation of the predicate in Listing 3 corresponds to what we needed for all the examples of this paper. For instance in Listing 1, the parameter of `new_promise` that was omitted was 0, to specify right to read and write on field `value` is transmitted alongside the reference used to resolve the promise. In the same listing, the pre-condition at line 5 shows the way we can specify what a method ensures about the result it provides through the resolution of a promise.

Next we comment on the encoding of the primitives for manipulating promises and futures: Listing 5 is the implementation of `new_promise`. It allocates the reference, and initialises all

```
predicate lock_inv(x: Ref){
  acc(x.fut_value, 1/2) &&
  acc(x.consumed, 1/2) &&
  x.consumed <= write &&
  x.consumed >= none &&
  acc(x.resolved, 1/2) &&
  (x.resolved ?
    acc(x.consumed, 1/2) &&
    acc(x.fut_value, wildcard) &&
    acc(future(x), x.consumed) &&
    acc(ip(x.fut_value, x), write - x.consumed) &&
    acc(x.resolved, wildcard)
  : true)
}
```

Listing 4: Viper Lock invariant

```
method new_promise(inv_number : Int) returns (res : Ref)
  ensures promise(res) && future(res)
  ensures unfolding future(res) in res.inv_id == inv_number
  ensures unfolding promise(res) in res.inv_id == inv_number
{
  res := new(locker, resolved, consumed, fut_value, inv_id)
  res.resolved := false
  res.consumed := none
  res.inv_id := inv_number
  UNLOCK(res)
  fold promise(res)
  fold future(res)
}
```

Listing 5: Viper promise creation implementation

the fields. It folds the lock invariant in its initial state, and unlocks it. Its last operation is to fold the two predicates ensured by this method: `promise()` and `future()`.

Listing 6 depicts method `resolve(pro, val)`. It requires predicate `promise(pro)` and the promise invariant `ip(val, pro)`. This method has no post-condition, so its calls consume permissions on the promise and the promise invariant. Notice how, thanks to the full ownership on fields `fut_value` and `resolved` provided by the combination of the lock invariant and the promise predicate, the values of those fields are updated, to the computed `v` and `true` respectively. Since the field `resolved` is now true, the ownership of the promise invariant has to be captured when folding the lock invariant. It also means that the fraction of permission on the field `consumed` that was part of the predicate `promise` is now available in the lock invariant, allowing the following get calls to update this field.

Since calls to `resolve` require the promise invariant `ip`, this predicate always has to be folded prior to resolving a promise. Because `ip` depends on `p.inv_id`, folding it requires access to this field. This is granted by unfolding the `promise(x)` predicate. The program state must

```
method resolve(x: Ref, v: Ref)
  requires promise(x)
  requires ip(v, x)
{
  LOCK(x)
  unfold lock_inv(x)
  x.resolved := true
  x.fut_value := v
  UNLOCK(x)
}
```

Listing 6: Viper resolve implementation

contain this predicate to resolve the promise anyway, but this operation needs to be done explicitly. To simplify writing programs using futures in Viper, we offer a macro doing these operation automatically. We can therefore simply write `RESOLVE(promise, value)`, as it was shown in Listing 1.

The implementation of the `get` primitive is a recursive active wait: if the future is resolved, the value is returned, otherwise get is recursively called with the same arguments. This implementation is shown in Listing 7. The presence of logical annotations renders the code more involved. Indeed, they take into account that the get primitive can be used both when the promise has still to be resolved, or when it is resolved. We therefore need to mimic a disjunction in both the pre and post conditions of this method.

To this aim, we add the parameter `quantity`, which represents the amount of permission on the `future(x)` predicate that will be consumed, and therefore the amount of permission that will be returned on the promise invariant. If this parameter is set to `none`, then we are encoding the case where there are multiple calls to `get` in the same thread. In this case the pre-condition is to have permission on the `resolved_future` predicate, ensuring this is indeed not the first call to `get`. As expected, no permission on the promise invariant is provided in this situation. Moreover, as we have to unfold the predicates manually, this disjunction appears in the code as well, as seen with the branching at line 12. Depending on the quantity parameter we know which permission was required, and therefore which predicate has to be unfolded. Finally, the disjunction also appears when folding the `future` predicate at line 24, but in a different way because here predicate `resolved_future` needs to be folded in both cases. Notice that a partial ownership (a wildcard permission) on `resolved_future` is always enough.

When the parameter `quantity` is different from `none`, some permission on the promise invariant must be released. This permission comes from the predicate `lock_inv` (Listing 4). Observe that when folding this predicate, the amount of permission on the promise invariant (`ip`) that is not captured back into the predicate is equal to the value that was added to the field `x.consumed`. This is also equal to the amount of permission that was captured on `future(x)`, because of line 10. This ensures that the amount of permission released on `ip(x.value, x)` cannot exceed the amount of permission that was held on `future(x)`.

The last post-condition of `get`, at line 9 ensures that the results of two consecutive gets are indeed the same memory reference. Because this is the way `get` will be used most of the time, we provide a `GET(f, res)` macro that corresponds to a `res := get(f, quantity)` statement with `quantity` being set to the amount of permission held on `future(f)` in the current program state. This macro also deals with unfolding the promise invariant if applicable, and as shown

```
method get(x: Ref, quantity: Perm) returns (v: Ref)
  requires quantity >= none && quantity <= write
  requires quantity > none ?
    acc(future(x), quantity)
  : acc(resolved_future(x), wildcard)
  ensures acc(ip(v, x), quantity)
  ensures quantity acc(resolved_future(x), wildcard)
  ensures unfolding acc(resolved_future(x), wildcard) in x.resolved
  ensures unfolding acc(resolved_future(x), wildcard)
    in x.resolved && v == x.fut_value
{
  if (quantity > none) {
    unfold acc(future(x), quantity)
  } else {
    unfold acc(resolved_future(x), wildcard)
  }
  LOCK(x)
  if (!x.resolved) {
    UNLOCK(x)
    fold acc(future(x), quantity)
    v := get(x, quantity)
  } else {
    v := x.fut_value
    if (quantity > none) {
      fold acc(future(x), quantity)
    }
    x.consumed := x.consumed + quantity
    assume x.consumed <= write
    UNLOCK(x)
    fold acc(resolved_future(x), quantity)
  }
}
```

Listing 7: Viper get implementation

in Listing 8, it would be cumbersome to have to write all those lines for each `get` call.

Finally, to implement asynchronous calls to methods, observe that we only need to consume the permission described by the pre-condition of the method we call, and the post-condition is empty. This means that if we can verify that 1) a method can run with its specified pre-condition 2) the asynchronous call can be done in a given environment, and 3) that all subsequent instructions do not need the resources consumed by this call, we know that the called method can run in parallel with the code that called it. The first and second point are checked automatically by the Viper infrastructure. The last point follows from the fact that, in our encoding, asynchronous methods do not have post-conditions. This implies that the resources mentioned in their pre-conditions will be removed from the environment from which they are called, and subsequent instructions will not be able to use permission on those resources.

4 Case Studies

```
define GET(fut, res) {
  var futperm: Perm
  futperm := perm(future(fut))
  unfold acc(future(fut), futperm)
  fold acc(future(fut), futperm)
  res := get(fut, perm(future(fut)))
  unfold acc(ip(res, fut), futperm)
}

define RESOLVE(p, x) {
  unfold promise(p)
  fold ip(x, p)
  fold promise(p)
  resolve(p, x)
}
```

Listing 8: Implementation of the macros simplifying get and resolve

```
method produce(n: Int, p: Ref)
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 1
{
  if (n == 0) {
    RESOLVE(p, null)
  }
  else {
    var cell: Ref
    cell := new(value, next)
    cell.value := n - 1
    var x: Ref
    x := new_promise(1)
    cell.next := x
    RESOLVE(p, cell)
    produce(n - 1, x)
  }
}
```

Listing 9: Producer/Consumer example: producer

To illustrate how our implementation works, we give two examples inspired by [4]. The first one is a producer consumer scheme, and the second a binary tree merging algorithm.

4.1 Producer Consumer

The producer provides a list, built cell by cell, and the consumer does a computation on the produced list. Here, the produced list is composed by integers from $n-1$ to 0, and the consumer simply sums up the values of all the cells. The cells produced differ from the typical list cells in the fact that the pointer to the next element is a future rather than an actual memory cell.

```
method consume(l: Ref, p: Ref, accu: Int)
  requires future(l)
  requires unfolding future(l) in l.inv_id == 1
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 0
{
  var cell: Ref
  GET(l, cell)
  if (cell == null) {
    var res: Ref
    res := new(value)
    res.value := accu
    RESOLVE(p, res)
  }
  else {
    consume(cell.next, p, cell.value + accu)
  }
}
```

Listing 10: Producer/Consumer example: consumer

```
method main() {
  var x: Ref
  x := new_promise(1)

  var p: Ref
  p := new_promise(0)

  consume(x, p, 0)
  produce(2, x)

  var res: Ref
  GET(p, res)
}
```

Listing 11: Producer/Consumer example: interaction between the two actors

This allows the consumer to run in parallel with the producer: the producer can return a cell as soon as its value is computed. The consumer can therefore compute a partial result for each cell as soon as it is produced. Viper code for the producer can be seen in Listing 9.

The implementation of the consumer we propose in Listing 10, illustrates one of the perks of having an explicit promise pointer to write the result of a future. Indeed, notice how the promise `p` that is used to collect the result of the computation of the consumer is passed through successive recursive calls and is only resolved by the last call. Using futures implicitly resolved with the termination of the asynchronous thread would lead to a result nested in as many futures as there were recursive calls.

Finally, Listing 11 shows how the producer and the consumer can work together. Notice that as mentioned in the previous section, because both `produce` and `consume` only have pre-

```
split(splitter, tree) :=
  if (tree == empty) then return empty
  else
    if (tree.root > splitter) then
      split_l, split_r := split(splitter, tree.left)
      return (split_l, new tree(tree.root, split_r, tree.right))
    else
      split_l, split_r := split(splitter, tree.right)
      return (new tree(tree.root, tree.left, split_l), split_r)

merge(t1, t2) :=
  if (t1 == empty) return t2
  elif (t2 == empty) return t1
  else
    split_l, split_r := split(t1.root, t2)
    return new tree(t1.root, merge(t1.left, split_l), merge(t1.right, split_r))
```

Listing 12: Tree merging algorithm

conditions, we can consider them as asynchronous methods. The order of the calls to those methods does not make any difference.

4.2 Tree Merging

The next example we implemented in Viper using promises and futures, is a binary search tree merging algorithm. As customary, in a binary search tree with root r, all the values in the left subtree are less than or equal to r, and all the values in the right subtree are greater than r. The algorithm merges two trees $t1$ and $t2$ into a new binary search tree. It is composed by two methods, *split* and *merge*. The method *split* takes a tree t and a value *splitter*, and returns two subtrees of t such that all values of t that were less than or equal to *splitter* are in the first subtree, and all values of t that were greater than *splitter* are in the second subtree. This method proceeds by recursively splitting the left or right subtree of t depending on whether r is greater than *splitter* or not. The method *merge* splits $t2$ using the root of $t1$ as splitter value. It then recursively merges the first half of the split to the left subtree of $t1$, and the second half to the right subtree of $t1$. The algorithm for those methods is shown in Listing 12.

The idea behind the introduction of futures in this algorithm is similar to the one of the previous example. Indeed, notice that the tree structure is recursive in the same way as the the list used in the producer consumer example. Each call of the merge algorithm builds one tree, its root is known from the arguments, and the subtrees are computed by the recursive calls. The intuition is to make the recursive calls asynchronous and to provide the tree computed at each step right away, delaying the subtrees as futures. In a similar way, all the recursive calls to method split can be done asynchronously. As the trees are split from top to bottom, their data will be accessed in the same order as the one of their computation.

To encode this example in our paradigm with explicit promises, we added one promise per return value for the two methods. The method *merge* has to instantiate two new promises before calling *split*, and another one before its recursive call. Notice that the method *split* only has to instantiate one new promise: in each case, it returns one of the results of its recursive call as it is. It means that this recursive call can be completely in charge of fulfilling one of the

```
field value: Ref
field left: Ref
field right: Ref

field type_of_tree: Int
/*  type_of_tree = 0 -> Empty
    type_of_tree = 1 -> Tree
    type_of_tree = 2 -> Fut of tree */

predicate tree(x: Ref){
  acc(x.type_of_tree) &&
  x.type_of_tree < 3 &&
  (x.type_of_tree == 1 ?          // We have an actual tree
      (acc(x.value) && acc(x.left) && acc(x.right) &&
      tree(x.left) && tree(x.right))
    : (x.type_of_tree == 2 ?      // We have a future on a tree
          future(x) &&
          unfolding future(x) in x.inv_id == 2
        : (x.type_of_tree == 0 ?  // We have an empty tree
              true
            : false)))
}
```

Listing 13: Tree predicate for the tree merging example

promises it had in its arguments. The full implementation in Viper of both methods of this example using promises is available at [1].

To represent the trees, we used references, and in addition to the `value` field that we had from the first example, representing here the root of a tree, we needed two fields to represent the left and right subtrees. We chose to represent the trees using an enumerated type, as it would be done in a functional language. A tree may be an empty tree; an actual tree with a value, a left, and right subtrees; or a future of a tree. To encode this type we used another field: `type_of_tree`. Its integer value for a given reference tells us which type of tree this reference is. The last element required to encode the trees is a predicate encapsulating the permissions on the fields of a tree reference. This predicate, alongside with the fields declaration, is shown in Listing 13. It is a parametric predicate depending on the value of the field `type_of_tree`. If the reference is an actual tree (immediately containing data, case where `type_of_tree` = 1), the predicate provides permission on all the fields composing the tree and ensures recursively that the left and right subtrees are trees as well. If the reference is an empty tree (`type_of_tree` = 0), the predicate does not provide any permission other than the one on field `type_of_tree`. If the reference is a future of a tree (`type_of_tree` = 2), the `tree` predicate ensures that the value this future will be resolved with, will be a tree. This is ensured by enforcing the field `inv_id` of the future to be 2, as we defined `ip(x, f)` to be `tree(x)` if `f.inv_id` is 2. Note that in this last case, the reference we deal with is a future on a tree and a tree itself. Not all futures on trees are themselves trees. For a reference to be considered as a tree, permission on its field `type_of_tree` must be available.

One interesting consideration is that, as our algorithm takes such trees as input, we have therefore to take into account that the input might be either an immediate tree or a future on

```
method split(splitter: Int, current_tree: Ref, pl: Ref, pr: Ref)
  requires tree(current_tree) && promise(pl) && promise(pr)
  requires unfolding promise(pl) in pl.inv_id == 2
  requires unfolding promise(pr) in pr.inv_id == 2
{
  unfold tree(current_tree)
  if (current_tree.type_of_tree == 0) {
    ...
  }
  else {
    if (current_tree.type_of_tree == 1) {
      ...
    } else {
      var actual_tree: Ref
      GET(current_tree, actual_tree)
      split(splitter, actual_tree, pl, pr)
    }
  }
}
```

Listing 14: Removing the future layers with recursive calls

a tree. Recursively, because a tree can be a future on a tree, the actual data of the tree (that is an empty tree or a root and a pair of pointers) may be nested in an arbitrary large number of future layers. We will show two ways we used to deal with this situation in the implementation of the two methods of the tree merging algorithm.

The first one is used in the implementation of the method *split*. We built this method around a disjunction on the value of the field `type_of_tree` of the input tree. The last case of this disjunction is illustrated in Listing 14, and is the case where the input tree is a future. The idea is to use a `get` call to wait for the future to be resolved and retrieve its value, and then to recursively call `split` again with this value. Because of the promise invariant we know that this value is a tree, and if it was again a future on a tree, the recursive call would again fall in the same case of the disjunction, triggering a new recursive call. The same idea would apply to this new recursive call, and so on until the value gotten from the future is not itself a future.

The approach we used in method merge differs because here there are two trees as input, and using the same disjunction is not ideal. Instead, we used a while loop to call `get` as many times as required on the trees given as argument. The implementation of this technique is shown in Listing 15. After the while loops, we know that the references `actual_t1` and `actual_t2` contain actual data, and are not futures.

5 Concluding Remarks

In this paper we have introduced the concept of promise invariant, allowing to reason about the resource transferred between asynchronous threads through promise resolution. We have proposed an implementation of the promise/future mechanism in Viper, based on this promise invariant. Two case studies showcasing how our Viper library can be used to verify actual programs conclude our presentation.

```
method merge(t1: Ref, t2: Ref, p: Ref)
  requires tree(t1) && tree(t2)
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 2
{
  var actual_t1: Ref
  var actual_t2: Ref

  actual_t1 := t1
  actual_t2 := t2

  while (unfolding tree(actual_t1) in actual_t1.type_of_tree == 2)
    invariant tree(actual_t1)
  {
    unfold tree(actual_t1)
    var temp: Ref
    GET(actual_t1, temp)
    actual_t1 := temp
  }

  while (unfolding tree(actual_t2) in actual_t2.type_of_tree == 2)
    invariant tree(actual_t2)
  {
    unfold tree(actual_t2)
    var temp: Ref
    GET(actual_t2, temp)
    actual_t2 := temp
  }
  ...
}
```

Listing 15: Removing the future layers with while loops

We conclude with some considerations about limitations of our current proposition and future works.

Implementation. One limitation of our implementation is that it does not rely on Viper verification to ensure that the quantity of permission on the promise invariant released by successive calls to `get` is not more than `write`. Instead, we need to use an `assume` primitive, as it is shown at line 28 in Listing 7. This primitive instructs the verification tool to "assume" the assertion is true, even if it cannot verify it.

Viper. Since Viper does not implement higher order predicates, the implementation of the promise invariant is not ideal. In fact we have to define the predicate `ip` in the same file as the promise/futures handling primitives. Because the definition of this predicate has to take into account all the different invariants needed to prove a program, the proof of the program has to be done in the same file as the implementation of the primitives, and we cannot provide a library working as a black box. This limitation also prevents us from implementing a logical connection between the arguments of a method and the resource invariant of a promise it resolves. For instance, in a simple program as the one in Listing 1, we cannot design a promise invariant allowing us to assert that `a.value` is equal to 2 times what `x.value` was before calling `m`.

Explicit and implicit futures. Terminology and implementation highly vary depending on the chosen programming language. In some languages, a future is created for each spawned thread, whereas the promise is implicitly filled by the return statement of the thread. In some other languages, the usage of futures is even more implicit: no explicit get instruction is needed to access the value stored in the future, and the compiler inserts the missing get when it guesses that it is needed. The respective advantages of explicit versus implicit futures is discussed in [9]. It would be interesting to explore how our logic could be adapted to deal with such implicit futures.

Data race freedom and absence of race condition. Separation logic is known to ensure data race freedom: in a proved program, all memory accesses are performed on owned locations. But still the program can have "race conditions" in the sense that its outcome can be non-deterministically depending on the scheduler. For some synchronisation primitives, like locks, data races and race conditions are two different things. But for some other synchronisation primitives, it can be expected that the ownership discipline imposed by separation logic enforces a form of determinism. This has been formalised for instance for channel communications in [11]. It would be interesting to see if a form of determinism can be enforced for a toy imperative, parallel programming language with pointers and futures.

Cost models and parallel time complexity. Blelloch and Reid-Miller argue in [4] that futures allow to give efficient parallel implementations of sequential algorithms with very light modifications; in their paper, they study the parallel time complexity of some algorithms based on future. This is one of these tree manipulating algorithms that we implemented and proved correct in Viper. It would be interesting to go further and also establish the parallel complexity of this algorithm. Two recent line of research are particularly attractive: separation-logic based proofs of (sequential) time complexity [13, 23], and type-based characterisations of parallel complexity [2, 3]. It would be very interesting to combine the ideas of these two lines of work and develop a proof system for parallel time complexity based on separation logic.

Formal proofs. This work is preliminary to a more foundational approach. We aim at proving formally the soundness of the axioms that are here encoded in Viper logic.

Acknowledgements. We would like to thank all the JFLA anonymous reviewers for their comments that greatly improved the present paper.

References

[1] Git repository of viper implementation. https://gitlab.com/lgermerie/viperfutures.

[2] Baillot, P., and Ghyselen, A. Types for complexity of parallel computation in pi-calculus. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings* (2021), pp. 59–86.

[3] Baillot, P., Ghyselen, A., and Kobayashi, N. Sized types with usages for parallel complexity of pi-calculus processes. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference* (2021), pp. 34:1–34:22.

[4] Blelloch, G. E., and Reid-Miller, M. Pipelining with futures. *Theory Comput. Syst. 32*, 3 (1999), 213–239.

[5] Bornat, R., Calcagno, C., O'Hearn, P. W., and Parkinson, M. J. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005* (2005), pp. 259–270.

[6] Denis, X. Mastering program verification using possession and prophecies. In *JFLA* (2021).

[7] Din, C. C., and Owe, O. Compositional reasoning about active objects with shared futures. *Formal Aspects Comput. 27*, 3 (2015), 551–572.

[8] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (2013), pp. 287–300.

[9] Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E. B., and Wrigstad, T. Godot: All the benefits of implicit and explicit futures. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom* (2019), pp. 2:1–2:28.

[10] Filliâtre, J., and Paskevich, A. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 125–128.

[11] Francalanza, A., Rathke, J., and Sassone, V. Permission-based separation logic for message-passing concurrency. *Log. Methods Comput. Sci. 7*, 3 (2011).

[12] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., and Sagiv, M. Local reasoning for storable locks and threads. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings* (2007), pp. 19–37.

[13] Guéneau, A., Charguéraud, A., and Pottier, F. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* (2018), pp. 533–560.

[14] Haack, C., Huisman, M., and Hurlin, C. Reasoning about java's reentrant locks. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings* (2008), pp. 171–187.

[15] Hinrichsen, J. K., Bengtson, J., and Krebbers, R. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang. 4*, POPL (2020), 6:1–6:30.

[16] Hobor, A., and Gherghina, C. Barriers in concurrent separation logic. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings* (2011), pp. 276–296.

[17] Hobor, A., and Gherghina, C. Barriers in concurrent separation logic: Now with tool support! *Log. Methods Comput. Sci. 8*, 2 (2012).

[18] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. Verifast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings* (2011), M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617 of *Lecture Notes in Computer Science*, Springer, pp. 41–55.

[19] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program. 28* (2018), e20.

[20] Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., and Jacobs, B. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang. 4*, POPL (2020), 45:1–45:32.

[21] Leino, K. R. M., Müller, P., and Smans, J. Deadlock-free channels and locks. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), pp. 407–426.

[22] Lozes, É., and Villard, J. Shared contract-obedient channels. *Sci. Comput. Program. 100* (2015), 28–60.

[23] Mével, G., Jourdan, J., and Pottier, F. Time credits and time receipts in iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings* (2019), pp. 3–29.

[24] Müller, P., Schwerhoff, M., and Summers, A. J. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, vol. 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security.* IOS Press, 2017, pp. 104–125.

[25] O'Hearn, P. W., Yang, H., and Reynolds, J. C. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004* (2004), pp. 268–280.

[26] Parkinson, M. J. The next 700 separation logics - (invited paper). In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings* (2010), pp. 169–182.

[27] Pottier, F. Hiding local state in direct style: A higher-order anti-frame rule. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA* (2008), pp. 331–340.

[28] Sampaio, G., Santos, J. F., Maksimovic, P., and Gardner, P. A trusted infrastructure for symbolic analysis of event-driven web applications. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)* (2020), pp. 28:1–28:29.

[29] Santos, J. F., Maksimovic, P., Ayoun, S., and Gardner, P. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (2020), pp. 927–942.

[30] Schwinghammer, J., Birkedal, L., Pottier, F., Reus, B., Støvring, K., and Yang, H. A step-indexed kripke model of hidden state. *Math. Struct. Comput. Sci. 23*, 1 (2013), 1–54.

[31] Vafeiadis, V., and Parkinson, M. J. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings* (2007), pp. 256–271.

[32] Villard, J., Lozes, É., and Calcagno, C. Proving copyless message passing. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings* (2009), pp. 194–209.