Futures and Promises in Haskell and Scala (Extended Version)

Tamino Dauth
Karlsruhe University of Applied Sciences
Karlsruhe, Germany

Martin Sulzmann
Karlsruhe University of Applied Sciences
Karlsruhe, Germany

ABSTRACT

Futures and promises are a high-level concurrency construct to aid the user in writing scalable and correct asynchronous programs. We introduce a simple core language based on which we can derive a rich set of future/promise features. We discuss ways to implement the core features via shared-state concurrency making either use of Software Transactional Memory, an elementary lock-based primitive, or an atomic compare-and-swap operation. The approach has been fully implemented in Haskell and Scala. For both languages, we provide empirical evidence of the effectiveness of our method. We consider program transformations in the context of futures/promises and observe potential problems in existing Scala-based libraries.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming languages**; **Concurrent programming structures**; **Domain specific languages**;

KEYWORDS

concurrency, futures, promises, EDSL, program transformation

1 INTRODUCTION

Futures and promises [1, 8] are a high-level concurrency construct to support asynchronous programming. A future can be viewed as a placeholder for a computation that will eventually become available. The term promise is often referred to a form of future where the result can be explicitly provided by the programmer. We consider here futures/promises supported via a library.

Our interest is to identify core features to support futures and promises and multiple ways to implement them based on the concurrency models available in the host language. We wish to derive a rich set of combinators and possible apply some program transformations to aid the effective execution of future/promises. Our source of inspiration are two powerful future/promise libraries. Scala's standard future/promise library (Scala FP) [9] and Twitter's own Scala future/promise library (Twitter FP) [6]. We largely follow their design principles but make several new contributions and provide new insights as we outline below.

There are of course numerous other languages, e.g. consider [2–4, 7, 11, 12, 16], that support some form of future/promise. We believe that Scala supports some of the most expressive future/promise libraries. Hence, our design follows Scala. We also consider Haskell because of its good support for various elementary shared-state concurrency primitives. Prior works [13, 17] in the Haskell setting are rather limited (in terms of the set of available combinators) compared to Scala.

, ,
2019.

This is work in progress where we report on our experiences implementing futures/promises in Haskell and Scala. In this paper, we make the following contributions:

- We formalize the semantics of a core language to support futures and promises (Section 2).
- We discuss several ways how to implement the core language based on shared-state concurrency making either use of Software Transactional Memory, an elementary lock-based primitive, or an atomic compare-and-swap operation (Section 3).
- We provide empirical results for the effectiveness of our implementations in Haskell and Scala (Section 4).
- We discuss some program transformations introduced by Scala-based libraries and observe that these transformations are unsound (Section 6).

The complete source code for Haskell and Scala can be accessed via [5, 18]. Due to space restrictions, we will only show the Haskell code in this paper. We assume a rudimentary understanding of the concept of futures and promises. In our view, at the semantic level there is no strict distinction between promises and futures. Hence, we largely use the term promise.

2 CORE LANGUAGE

We introduce a core language to express promises.

Definition 2.1 (Core Syntax).

p, q, r			Promise
v			Values
h, f, g			Callback
hs	::=	$[h_1, \ldots, h_n]$	Callback List
c	::=	p = new | tryComplete $p\ v$	
	|	onComplete $p\ h$ | get p	Commands

For brevity, we leave the language of values and callbacks abstract. We write $h(v)$ to denote the application of callback h on value v. The result of this computation does not matter at this point. We use Haskell style notation to denote a list of callbacks. Sometimes, we write $h : hs$ to denote a list with head element h and tail hs.

There are four commands in our core language. Via new we create a promise which we can try to complete via some value v via tryComplete $p\ v$. Command onComplete $p\ h$ registers a callback h which will be executed once the promise is completed. Via get we can wait for a promise to complete. In our simplified language, the value obtained by a completed promise is ignored.

We explain the semantics of our core language in terms of a structural operational semantics (SOS) defined by a set of rewrite rules. The rewrite rules operate on configurations $(\mathcal{P} \mid cs)$ and $(\mathcal{P} \mid C)$. The syntax of the individual components is as follows.

Definition 2.2 (Semantic Configurations).

cs	::=	$[c_1, \ldots, c_n]$	Thread
δ	::=	$v \mid hs$	Promise State
$\mathcal{P}$	::=	$\{\} \mid \{p_\delta\} \mid \mathcal{P} \cup \mathcal{P}$	Environment
C	::=	$\{\{\}\} \mid \{\{[h(v)]\}\} \mid \{\{cs\}\} \mid C \cup C$	Program

A thread cs is represented as a list of commands. There is no explicit command in our core language to create a new thread, for brevity. Creation of threads is implicit as we assume that the execution of callbacks will not block the main thread. We use multi-set notation $\{\{[h_1(v)], \ldots, [h_n(v)], cs\}\}$ to denote concurrent execution of call backs $h_i(v)$ and main thread cs.

A promise p can be in two states. Either the promise has been completed by a value v, denoted by p_v, or the promise has not been completed yet and some callbacks are waiting for execution, denoted by p_{hs}. The current state of each promise is recorded in some environment $\mathcal{P}$. We use set notation for environments. Hence, when writing $\{p_\delta\} \cup \mathcal{P}$ we can assume that promise p does not appear in $\mathcal{P}$.

Next, we introduce the SOS rewrite rules for commands executing in some environment. Each rewrite rule operates on some configuration $(\mathcal{P} \mid cs)$ and observes the leading command and if necessary the environment. As callbacks are executed concurrently to the main thread, the resulting configuration is of shape $(\mathcal{P} \mid C)$.

Definition 2.3 (Command Semantics $(\mathcal{P} \mid cs) \Rightarrow (\mathcal{P} \mid C)$).

$$\text{(New)} \quad (\mathcal{P} \mid p = \mathsf{new} : cs) \Rightarrow (\{p_{[]}\} \cup \mathcal{P} \mid \{\{cs\}\})$$

$$\text{(On-1)} \quad (\{p_{hs}\} \cup \mathcal{P} \mid \mathsf{onComplete}\ p\ h : cs) \Rightarrow (\{p_{h:hs}\} \cup \mathcal{P} \mid \{\{cs\}\})$$

$$\text{(On-2)} \quad (\{p_v\} \cup \mathcal{P} \mid \mathsf{onComplete}\ p\ h : cs) \Rightarrow (\{p_v\} \cup \mathcal{P} \mid \{\{[h(v)], cs\}\})$$

$$\text{(Try-1)} \quad (\{p_{[h_1,\ldots,h_n]}\} \cup \mathcal{P} \mid \mathsf{tryComplete}\ p\ v : cs) \Rightarrow (\{p_v\} \cup \mathcal{P} \mid \{\{[h_1(v)], \ldots, [h_n(v)], cs\}\})$$

$$\text{(Try-2)} \quad (\{p_v\} \cup \mathcal{P} \mid \mathsf{tryComplete}\ p\ v' : cs) \Rightarrow (\{p_v\} \cup \mathcal{P} \mid \{\{cs\}\})$$

$$\text{(Get)} \quad (\{p_v\} \cup \mathcal{P} \mid \mathsf{get}\ p : cs) \Rightarrow (\{p_v\} \cup \mathcal{P} \mid \{\{cs\}\})$$

Rule (New) creates a new promise. The state $p_{[]}$ indicates that the promise is not yet completed and no callbacks have been registered so far. For command onComplete $p\ h$ there are two cases to consider. If the promise is not yet completed, we simply add h to the list of callbacks. See rule (On-1). Otherwise, we execute the callback on the given value in some newly created thread. See rule (On-2). If promise p is not completed yet, command tryComplete $p\ v$ sets the promise state to p_v and executes all callbacks in each own thread. See rule (Try-1). Once a promise is completed its value cannot be overridden. See rule (Try-2). Command get p blocks until the promise is completed. See rule (Get).

A program is executed by executing one command after the other where the scheduling among threads can be arbitrary.

```
class Core t where
  newC :: IO (t a)
  getC :: t a -> IO (Maybe a)
  tryCompleteC :: t a -> IO (Maybe a) -> IO Bool
  onCompleteC :: t a -> (Maybe a -> IO ()) -> IO ()
```

Table 1: Core Features Interface

Definition 2.4 (Program Semantics $(\mathcal{P} \mid C) \Rightarrow (\mathcal{P} \mid C)$).

$$\text{(Step)} \quad \frac{(\mathcal{P} \mid cs) \Rightarrow (\mathcal{P}' \mid C')}{(\mathcal{P} \mid \{cs\} \cup C) \Rightarrow (\mathcal{P}' \mid C' \cup C)}$$

$$\text{(Closure)} \quad \frac{(\mathcal{P} \mid C) \Rightarrow (\mathcal{P}' \mid C') \quad (\mathcal{P}' \mid C') \Rightarrow (\mathcal{P}'' \mid C'')}{(\mathcal{P} \mid C) \Rightarrow (\mathcal{P}'' \mid C'')}$$

Based on the four operations in our core language, it is possible to derive a powerful library to support a rich set of future/promise features. The design is not restricted to a specific programming language assuming the core features can be provided. For concreteness, we will discuss an implementation in Haskell.

3 CORE HASKELL IMPLEMENTATION

Table 1 gives a description of the interface of the four core operations. We use a constructor class to abstract over their concrete implementation. Method `newC` yields a new promise of type `t a`. As creation relies on some side-effects, we assume that `newC` is carried out within the `IO` monad. The same applies to the other three methods. Completion of a promise may fail in the sense that we may encounter some exceptional case. This could be dealt with by raising an exception. For brevity, we assume that such side effects are represented by the `Maybe` data type.

There exist numerous ways to implement these core features. We consider methods based on shared-state concurrency. Haskell provides three of such such elementary synchronization primitives. Software Transactional Memory (STM) [10], an elementary lock-based primitive (MVar) [14], and an atomic compare-and-swap operation (CAS). In Haskell, CAS is supported via mutuable reference variables (IORef).

3.1 Shared-State Synchronization Primitives

Table 2 summarizes the main functions to access STM, MVar and IORef in Haskell. We first consider STM.

A STM *transaction* is a sequence of reads and writes performed on transactional variables (`TVars`). Transactions are executed *atomically*. The common implementation strategy to guarantee this property is based on *optimistic concurrency*. Transactions run to completion under the assumption that no conflicts have occurred. At the end of the transaction we check if any of the read (transactional) variables have changed. If yes the transactions is retried. That is, we re-start the transaction again canceling out any of the changes we made. Otherwise, all of writes are committed to memory and the transaction terminates. It is possible for the user to abort (retry) a transaction via the `retry` command. In this case, the transaction will only be re-started once any of the read variables have changed.

```
atomically :: STM a -> IO a
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
retry      :: STM ()

newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a

newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO a
casIORef   :: Eq a => IORef a -> a -> a -> IO Bool
```

Table 2: STM, MVar and IORef Haskell Interface

An MVar is like a one-place channel that can either be full or empty. Operation `takeMVar` blocks if the MVar is empty. Otherwise, it returns the value and the MVar becomes empty. Similarly, `putMVar` blocks if the MVar is full. Otherwise, the MVar becomes full by the value provided. Operation `readMVar` atomically reads the value of an MVar. In our setting, we can view `readMVar` as the composition of `takeMVar` and `putMVar`.

An IORef is a mutuable variable where the read and write IORef operations resemble `putMVar` and `readMVar`. The difference is that access to an IORef is not protected. Hence, data races are possible. To avoid such conflicts we find an atomic compare-and-swap operation `casIORef`. [1] The call `casIORef ref old new` compares the current value accessed via the reference `ref` against the value `old`. If equal, the current value is changed to the value `new`. Otherwise, no changes are made. The Boolean flag indicates if a change took place.

3.2 STM Implementation

Table 3 shows an implementation with STM. We use a transactional variable to represent the shared state of a promise. The either data type (we use our own for clarity) represents the possible (exclusive) states a promise can be in.

Method `newC` creates a new transactional variable where the initial value `R []` indicates that the promise is not yet completed and no callbacks have been registered so far. In method `getC` we atomically access the current state of a promise. If the promise is completed, see case "`L`", we return the value. Otherwise, we call STM's `retry`. That is, we effectively block the thread until the promise has been completed.

Method `tryCompleteC` attempts to complete a promise. We first query the current state of the promise. If not completed yet, see case "`R`", we change the state of the promise, and retrieve the list of callbacks for execution. This case corresponds to the operational semantics rule (Try-1). Querying and updating the state of the promise is done atomically. This guarantees consistency. For processing of callbacks, we build an action that makes use of `forkIO` to execute the callbacks without blocking the main thread. This action

[1] Haskell provides a more general `atomicModifyIORef` operation via which we can implement `casIORef`. The details are standard and can be found elsewhere [19].

```
data E a b = L a | R b
data CSTM a = CSTM (TVar (E (Maybe a) [Maybe a -> IO ()]))

instance Core CSTM where
  newC = do x <- atomically $ newTVar (R [])
            return $ CSTM x

  getC (CSTM x) = atomically $ do
                    s <- readTVar x
                    case s of
                      R _ -> retry
                      L v -> return v

  tryCompleteC (CSTM x) m = do
    v <- m
    action <- atomically $ do
                s <- readTVar x
                case s of
                  R hs ->
                    do writeTVar x (L v)
                       return $ do mapM_ (\h -> forkIO $ h v) hs
                                   return True
                  L _ -> return (return False)
    action

  onCompleteC (CSTM x) h = do
    action <- atomically $ do
                s <- readTVar x
                case s of
                  R hs -> do writeTVar x $ R (h:hs)
                             return (return ())
                  L v -> return (do forkIO $ h v
                                    return())
    action
```

Table 3: Core with STM

is executed outside of the STM transaction, after all operations in the STM transactions could commit. Case "`L`" covers the operational semantics rule (Try-2). As the promise is already completed, completion is not successful, indicated by returning the value false.

Similarly, the cases in method `onCompleteC` covers the operational semantic rules (On-1) and (On-2). We either register the callback if the promise is not completed yet, or simply retrieve its value. Like in case "`R`" of `tryCompleteC`, execution of the callback is performed outside of the STM transaction.

3.3 MVar Implementation

Table 4 gives an implementation of the core features using MVar. Instead of a transactional variable, we use an MVar to protect the (shared) state of a promise. To avoid spinning until the promise is completed, we introduce another MVar (of unit type) to signal any waiting getter.

Method `newC` creates an MVar promise with an empty list of callbacks. The other MVar is still empty. Operation `getC` blocks until that MVar becomes full and then retrieves the value of the completed promise.

The MVar implementation of method `tryCompleteC` accesses the promise state via `takeMVar`. If the promise is already completed,

```
data CMVAR a = CMVAR (MVar (E (Maybe a) [Maybe a -> IO ()]))
                     (MVar ())

instance Core CMVAR where
   newC = do x <- newMVar (R [])
             y <- newEmptyMVar
             return $ CMVAR x y

   getC (CMVAR x y) = do _ <- readMVar y
                         (L v) <- readMVar x
                         return v
   tryCompleteC (CMVAR x y) m = do
        v <- m
        s <- takeMVar x
        case s of
          L _ -> do putMVar x s
                    return False
          R hs -> do putMVar x (L v)
                     putMVar y ()
                     mapM_ (\h -> forkIO $ h v) hs
                     return True

   onCompleteC (CMVAR x y) h = do
       s <- takeMVar x
       case s of
         (L v) -> do putMVar x (L v)
                     forkIO $ h v
                     return ()
         (R hs) -> putMVar x (R (h:hs))
```

Table 4: Core with MVar

see case "L", we put back the taken value. Otherwise, we complete the promise and fill the other MVar to inform any waiting `getC` calls. Each callback is executed concurrently via `forkIO`. Consider operation `onCompleteC`. We attempt exclusive access to the state of the promise via `takeMVar`. If the promise is not completed yet, the callback is added to the list of pending callbacks. Otherwise, we put back the value and execute the callback.

3.4 CAS Implementation

The CAS implementation in Table 5 employs a spinning loop in combination with `casIORef` to consistently access and update the state of a promise. The call to `casIORef` requires to compare the current with the old value. In Haskell terms this means that we must provide an instance of the `Eq` type class on type `(E (Maybe a) [Maybe a -> IO ()])`. As callbacks are only added and once a promise is completed the value cannot be changed, the definition of `Eq` does not need to inspect actual values. See cases `CIO-1` and `CIO-2` in Table 5. We make use of an initially empty MVar to signal any waiting `getC` call once the promise is completed.

We briefly compare all three implementations. Compared to the STM and CAS implementation, the MVar implementation is less optimistic as any other (concurrent) access to the promise is blocked. In case of STM, we make use of `retry` to wait for a promise to complete whereas CAS requires an additional MVar for synchronization.

```
data CIO a = CIO (IORef (E (Maybe a) [Maybe a -> IO ()]))
                 (MVar ())

instance Eq (E (Maybe a) [Maybe a -> IO ()]) where
  (==) (L _) (R _) = False
  (==) (R _) (L _) = False
  (==) (L Nothing) (L (Just _)) = False
  (==) (L (Just _)) (L Nothing) = False
  (==) (L (Just _)) (L (Just _)) = True          -- CIO-1
  (==) (L Nothing) (L Nothing) = True
  (==) (R xs) (R ys) = length xs == length ys    -- CIO-2

instance Core CIO where
  newC =
    do x <- newIORef (R [])
       y <- newEmptyMVar
       return $ CIO x y

  getC (CIO x y) =
    do _ <- readMVar y
       (L v) <- readIORef x
       return v

  tryCompleteC (CIO x y) m = do
    v <- m
    let go = do
         val <- readIORef x
         case val of
           L _ -> return False
           R hs -> do b <- casIORef x val (L v)
                      if b
                       then do putMVar y ()
                               mapM_ (\h -> forkIO $ h v) hs
                               return True
                       else go
    go

  onCompleteC (CIO x _) h = do
       let go = do val <- readIORef x
                   case val of
                     L v -> do forkIO $ h v
                               return ()
                     R hs -> do b <- casIORef x val (R (h:hs))
                                if b
                                  then return ()
                                  else go
       go
```

Table 5: Core with CIO

4 EMPIRICAL RESULTS

We compare the performance among the STM, MVar and CAS versions for Haskell [18]. and Scala. Implementations for Scala follow a similar pattern and can be found here [5]. In addition, we provide a comparison against Scala and Twitter FP which both use CAS. For benchmarking we use four test cases. Haskell code snippets of benchmarks are shown in Tables 6, 7 and 8.

The first benchmark (Table 6) covers two test cases and measures high and low contention of promise callbacks and completions. The

```
-- high contention
let n = 10000     -- promises
let m = 100       -- onCompletes per promise
let k = 200       -- tryCompletes per promise
{- low contention
let n = 100000    -- promises
let m = 2  0      -- onCompletes per promise
let k = 2         -- tryCompletes per promise -}

ps <- mapM (\_ -> newC) [1..n]
let cmpls = mapM_ (\p -> onCompleteC p (\_ -> incCounter)) ps
let tries = mapM_ (\p -> tryCompleteC p (return $ Just 1)) ps

mapM_ (\_ -> forkIO cmpls) [1..m]
mapM_ (\_ -> forkIO tries) [1..k]
mapM_ getC ps
await (n*m)
```

Table 6: Test 1: High vs low contention

```
let n = 2000000 -- promises
ps <- mapM (\_ -> newC) [1..n]
let go (p1:p2:ps) =
       onCompleteC p1
                   (\_ -> do tryCompleteC p2 (return $ Just 1)
                             go (p2:ps))
    go [p] = onCompleteC p (\_ -> return ())
go ps
tryCompleteC (head ps) (return $ Just 1)
getC (last ps)
return ()
```

Table 7: Test 2: Chain of nested onCompletes and tryCompletes

benchmark creates n empty promises. For each promise, we register m callbacks and attempt to complete each k times. We then wait for all promises to be completed (`getC`) and all callbacks to be processed (global counter). Each promise executes its sequence of `onCompleteC` and `tryCompleteC` concurrently. In the Scala variant, we use an executor to manage the number of tasks based on the number of available threads. Each callback is submitted to the specified executor once the promise is completed. This is unnecessary in Haskell as Haskell supports light-weight threads. So, we simply use `forkIO`.

The second benchmark (Table 7) registers a nested chain of `onCompleteC` and `tryCompleteC` calls for a list of n promises. We complete the first promise and wait for completion of the last promise.

The third benchmark (Table 8) is a variant of the second benchmark. Instead of a deeply nested chain of callbacks and tries, we built up a flat sequence of `onCompleteC` and `tryCompleteC` calls.

For benchmarking we use a machine with 8 virtual and 4 physical Intel(R) Core(TM) i7-7700HQ CPUs @ 2.80GHz. We are using GHC 8.0.2 and Scala 2.12.7 on Fedora. Each benchmark is executed 10 times and we report the average execution time as the final result. For Scala, JIT is disabled. For Haskell, we report on the y-axis the average time measured in seconds and on the x-axis the number of

```
let n = 2000000 -- promises
ps <- mapM (\_ -> newC) [1..n]
let go (p1:p2:ps) = do
       onCompleteC p1
                   (\_ -> do tryCompleteC p2 (return $ Just 1)
                             return ())
       go (p2:ps)
    go [p] = onCompleteC p (\_ -> return ())
go ps
tryCompleteC (head ps) (return $ Just 1)
getC (last ps)
return ()
```

Table 8: Test 3: Chain of onComplete and tryComplete

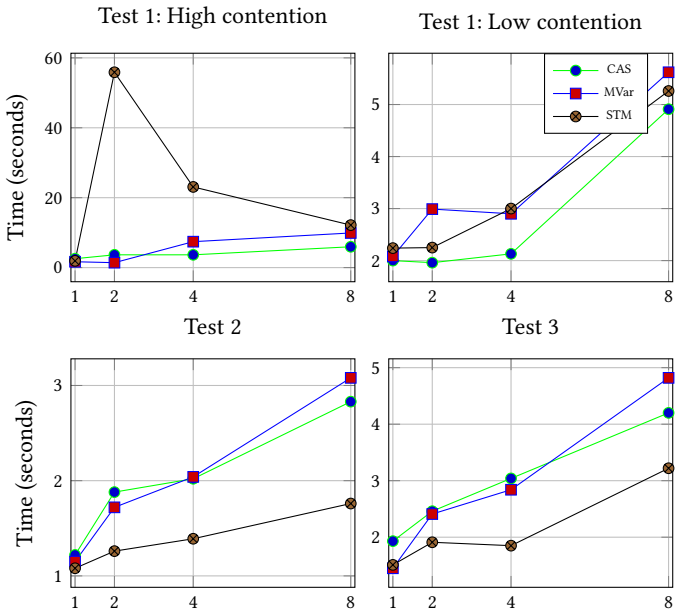

Figure 1: Benchmarks in Haskell

CPU cores used. For Scala, the x-axis reports the number of executor threads. Benchmark results for Haskell are shown in Figure 1 and for Scala in Figure 2.

For Haskell, the STM implementation performs best. For all Haskell implementations (STM, MVar and CAS), there is a degrade in performance with an increasing number of CPU cores. The reason might be some scheduling overhead but further investigation is necessary. The Scala implementations use one executor with a fixed number of threads instead. This might produce less scheduling for the threads and therefore performs better. For all Haskell benchmarks, fewer CPU cores/threads seem to improve the performance. This appears to be due to the small amount of concurrency in our test cases.

For Scala, STM appears to be slower compared to CAS and MVar. CAS is ahead of MVar, except for the low contention variant of the first benchmark. For the second and third benchmark, Scala FP shows the best performance, however, our implementations still have an acceptable execution time. Our CAS implementation is

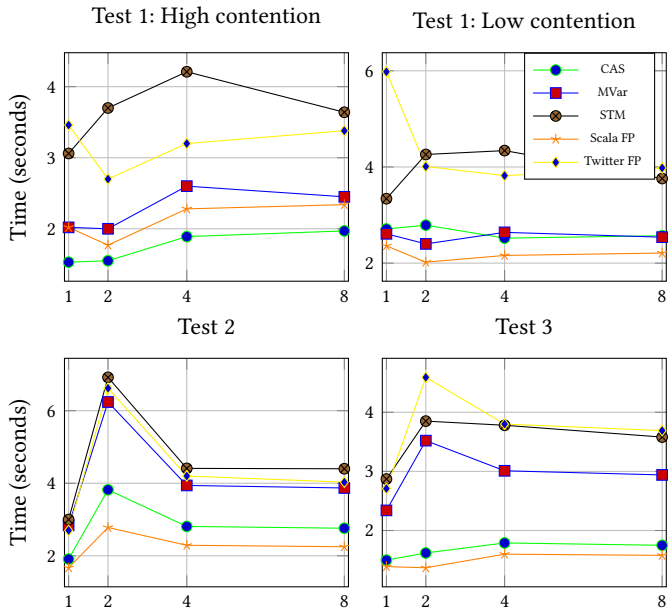

Figure 2: Benchmarks in Scala

```
class Future t where
  future :: IO (Maybe a) -> IO (t a)
  get :: t a -> IO (Maybe a)
  onComplete :: t a -> (Maybe a -> IO ()) -> IO ()
  onSuccess :: t a -> (a -> IO ()) -> IO ()
  onFail :: t a -> (() -> IO ()) -> IO ()

instance Core t => Future t where
  future m = do p <- newC
                forkIO $ do tryCompleteC p m
                            return ()
                return p
  get = getC
  onComplete f h = onCompleteC f h
  onSuccess f h = onCompleteC f (\x -> case x of
                                         Nothing -> return ()
                                         Just v -> h v)
  onFail f h = onCompleteC f (\x -> case x of
                                         Nothing -> h ()
                                         Just v -> return ())
```

Table 9: Future

ahead of Scala FP for the first benchmark. In this case, we encounter a fairly high number of callbacks. In our implementations, we have optimized registration of callbacks (`onCompleteC`) by avoiding to copy the entire list of existing callbacks. This seems to make a difference as Scala FP still uses copying of callbacks. [2] Twitter FP shows the worst performance for this case. The reason seems to be that submission of tasks (callbacks) to the executor is rather slow.

5 DERIVED FEATURES

Table 9 shows a set of features which are commonly attributed to futures. Futures cannot be explicitly tried. Rather, there is a method `future` to carry out a non-blocking computation by making use of `forkIO`. The computation is represented by a (core) promise and can either be accessed via the blocking `get` method, or variants of the non-blocking `onComplete` operation.

Table 10 introduces an extension of the `tryComplete` method where the value used for completion is represented by some (possibly not yet available) computation. In Scala FP, for

```
tryCompleteWith p q
```

we find that `p` and `q` must be different entities (of a different type). That is, `p` is of type `Promise a` and `q` is of type `Future a`. We do not make such a distinction here.

```
class FP t where
  trySuccWith :: t a -> t a -> IO ()
  tryFailWith :: t a -> t a -> IO ()
  tryCompleteWith :: t a -> t a -> IO ()

instance Core t => FP t where
  trySuccWith p f =
    onSuccess f (\x -> do trySuccess p x
                          return ())
  tryFailWith p f =
    onFail f (\() -> do tryFail p
                        return ())
  tryCompleteWith p q = do
    onCompleteC q (\x -> do tryCompleteC p (return x)
                            return ())
```

Table 10: Future/Promise

6 PROGRAM TRANSFORMATIONS

We have examined existing implementations of futures/promises for (run-time) program transformations. In case of the Scala programming language, we discovered that the standard future/promise library (Scala FP) [9] and Twitter's own future/promise library (Twitter FP) [6] apply some run-time transformation involving the `tryCompleteWith` operation. In case of Twitter FP, the use of the transformation is unrestricted and therefore leads to semantic incorrectness as we will explain shortly. In case of Scala FP, the transformation is restricted. Thanks to the fact that Scala FP distinguishes among futures and promises, the transformation appears to be semantically correct. To allow for a concise explanation of the issue, we extend our core language with the `tryCompleteWith` operation.

We add the following rules to the operational semantic rules in Definition 2.3

[2] The latest Scala version 2.13 seems to incorporate this optimization, so likely performance of Scala FP will improve.

Definition 6.1 (Semantics tryCompleteWith $p\ q$*).*

$$\text{(Try-W1)}\quad (\{p_{[h_1,\ldots,h_n]}, q_v\} \cup \mathcal{P} \mid \text{tryCompleteWith } p\ q : cs) \Rightarrow (\{p_v, q_v\} \cup \mathcal{P} \mid \{\![[h_1(v)], \ldots, [h_n(v)], cs]\!\})$$

$$\text{(Try-W2)}\quad (\{p_{hs}, q_{gs}\} \cup \mathcal{P} \mid \text{tryCompleteWith } p\ q : cs) \Rightarrow (\{p_{hs}, q_{gs}\} \cup \mathcal{P} \mid \{\![[\text{tryCompleteWith } p\ q], cs]\!\})$$

$$\text{(Try-W3)}\quad (\{p_v\} \cup \mathcal{P} \mid \text{tryCompleteWith } p\ q : cs) \Rightarrow (\{p_v\} \cup \mathcal{P} \mid \{\![cs]\!\})$$

In rule (Try-W1) we assume that the value of q is already available. This then leads to the execution of the callbacks connected to p. In rule (Try-W2) neither p nor q have been completed yet. As the call to tryCompleteWith $p\ q$ is non-blocking, we move the operation to its own thread. Rule (Try-W3) covers the case that p has already been completed.

The Twitter FP library applies a transformation where in case both p and q have not been completed yet, q's callbacks are moved to p. In terms of our semantic rules, we can describe this transformation by replacing rule (Try-W2) by the following rule.

Definition 6.2.

$$\text{(Try-W2b)}\quad (\{p_{hs}, q_{gs}\} \cup \mathcal{P} \mid \text{tryCompleteWith } p\ q : cs) \Rightarrow (\{p_{hs\ +\!+\ gs}, q_{[]}\} \cup \mathcal{P} \mid \{\![[\text{tryCompleteWith } p\ q], cs]\!\})$$

where ++ denotes concatenation among lists.

The intention of this transformation is to remove intermediate promises in case there is a chain of tryCompleteWith calls.

Suppose we reach the following program state

$$(\{p_{hs}, q_{fs}, r_{gs}\} \mid \{\![[\text{tryCompleteWith } p\ q], [\text{tryCompleteWith } p\ r], [\text{tryComplete } q\ v]]\!\}$$

Via two consecutive applications of the new rule (Try-W2b) applied on tryCompleteWith $p\ q$ and then on tryCompleteWith $p\ r$ we reach

$$(\{p_{hs\ +\!+\ fs\ +\!+\ gs}, q_{[]}, r_{[]}\} \mid \{\![[\text{tryCompleteWith } p\ q], [\text{tryCompleteWith } p\ r], [\text{tryComplete } q\ v]]\!\}$$

Application of rule (Try-1) on tryComplete $q\ v$ yields

$$(\{p_{hs\ +\!+\ fs\ +\!+\ gs}, q_v, r_{[]}\} \mid \{\![[\text{tryCompleteWith } p\ q], [\text{tryCompleteWith } p\ r]]\!\}$$

We are in the position to apply rule (Try-W1) and obtain

$$(\{p_v, q_v, r_{[]}\} \mid \{\![[k_1(v)], \ldots, [k_n(v)], [\text{tryCompleteWith } p\ r], cs']\!\})$$

where hs ++ fs ++ $gs = [k_1, \ldots, k_n]$.

We encounter some semantic incorrectness! The callbacks of r are executed with q's value and there is no guarantee that r will ever be completed. This (incorrect) behavior is due to rule (Try-W2b) and cannot be reproduced with the rules in Definition 2.3 and 6.1.

The above (incorrect) behavior can be reproduced in Twitter FP. For details, see here [5].

Scala FP applies a similar transformation but the transformation is restricted. In a chain of `tryCompleteWith` calls [3], the intermediate p's introduced are locally bound. Further, the resulting 'promise' has the type `Future a`. As a future cannot be 'tried' there is the guarantee that applications of rule (Try-W2) are restricted to calls tryCompleteWith $p\ q$ where there is no other competing tryCompleteWith $p\ r$. Hence, the above example does not immediately apply to Scala FP. However, it is possible to cast a future to promise and then a variant of the above example also applies to Scala FP.

In future work, we will establish some concise conditions under which rule (Try-W2) is sound. We will examine further transformation cases and their impact on the performance of programs.

ACKNOWLEDGMENTS

We thank the PEPM'19 reviewers for their comments.

REFERENCES

[1] Henry C. Baker, Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. *SIGPLAN Not.* 12, 8 (Aug. 1977), 55–59.
[2] C# 2018. C# Tasks. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=netframework-4.7.2. (2018).
[3] Clojure 2018. Clojure Futures. https://clojuredocs.org/clojure.core/future. (2018).
[4] Alex Crichton. 2018. Rust Zero-Cost Futures. https://docs.rs/futures/. (2018).
[5] Tamino Dauth. 2018. Futures and Promises in Haskell and Scala - Scala Source Code. https://github.com/tdauth/PEPM19-supplementary-material-scala. (2018).
[6] Finagle 2018. Concurrent Programming with Futures. https://twitter.github.io/finagle/guide/Futures.html. (2018).
[7] Folly 2018. Folly: Facebook Open-source Library. https://github.com/facebook/folly. (2018).
[8] Daniel Friedman and David Wise. 1976. The Impact of Applicative Programming on Multiprocessing. (1976), 263–272 pages.
[9] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2018. Futures and Promises. http://docs.scala-lang.org/overviews/core/futures.html. (2018).
[10] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. 2005. Composable memory transactions. In *Proc. of PPoPP'05*. ACM Press, 48–60.
[11] Java 2018. Java Futures. https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Future.html. (2018).
[12] JavaScript 2015. JavaScript Promises. http://www.ecma-international.org/ecma-262/6.0/index.html#sec-promise-objects. (2015).
[13] Chris Kuklewic. 2009. future: Supposed to mimics and enhance proposed C++ future features. http://hackage.haskell.org/package/future. (2009).
[14] S. Peyton Jones, A. Gordon, and S. Finne. 1996. Concurrent Haskell. In *Proc. of POPL'96*. ACM Press, 295–308.
[15] John H. Reppy. 1999. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA.
[16] Saarland University 2007. The Alice Manual. http://www.ps.uni-sb.de/alice/manual/. (2007).
[17] David Sabel. 2009. caf: A library of Concurrency Abstractions using Futures. http://hackage.haskell.org/package/caf. (2009).
[18] Martin Sulzmann. 2018. Futures and Promises in Haskell and Scala - Haskell Source Code. https://github.com/sulzmann/PEPM19-supplementary-material. (2018).
[19] Martin Sulzmann, Edmund S. L. Lam, and Simon Marlow. 2009. Comparing the performance of concurrent linked-list implementations in Haskell. In *Proc. of DAMP'09*. ACM, 37–46.

A ATOMIC COMPLETION

Table 11 introduces a combinator `tryCompleteAtomicAll` to atomically complete a list of promises. That is, we complete all or none. We exploit this feature to implement a non-deadlocking version of the dining philosophers problem. In a more complex application we show how to derive a library-based implementation of

[3] In Scala FP, the transformation does not directly operate on `tryCompleteWith`. Instead, the transformation only applies to `flatMap` but the effects are similar.

```
tryCompleteAtomicAll :: [(CSTM t, Maybe t)] -> IO Bool
tryCompleteAtomicAll xys = do
  let unPrimSTM (CSTM x) = x
  let xs = map (unPrimSTM . fst) xys
  let ys = map snd xys
  let zys = zip xs ys
  action <- atomically $ do
    (let go [] a1 a2 = do a1
                          return $ do a2
                                      return True
         go ((z,y):pys) a1 a2 = do
            p <- readTVar z
            case p of
              R hs -> go pys
                         (do a1
                             writeTVar z (L  y))
                         (do a2
                             forkIO_ $ mapM_ (\h -> h y) hs)
              L _ -> retry
     in go zys (return ()) (return ()))
    `orElse`
    (return (return False))
  action
```

Table 11: Atomically complete

```
class Combinators t where
  transformWith :: t a -> (Maybe a -> IO (t b)) -> IO (t b)
  transform :: t a -> (Maybe a -> IO (Maybe b)) -> IO (t b)
  guard :: t a -> (a -> IO Bool) -> IO (t a)
  orAlt :: t a -> t a -> IO (t a)
  first :: t a -> t a -> IO (t a)
  firstSucc :: t a -> t a -> IO (t a)
```

Table 12: Combinators

```
instance Core t => Combinators t where
transformWith f h = do
  p <- new
  onComplete  f (\x -> do v <- h x
                          tryCompleteWith p v)
  return p
transform f h = do
  transformWith f (\x -> do p <- new
                            tryComplete p (h x)
                            return p)
guard f h = followedBy f (\x -> do v <- h x
                                   if v
                                      then return $ Just x
                                      else return Nothing)

orAlt f1 f2 = do
  transformWith f1
    (\x -> case x of
             Just v -> return f1
             Nothing -> transform f2
                          (\x -> case x of
                                   Just v -> return x
                                   Nothing -> return Nothing))
first f1 f2 = do
  p <- new
  tryCompleteWith p f1
  tryCompleteWith p f2
  return p
firstSucc f1 f2 = do
  p <- new
  trySuccWith p f1
  trySuccWith p f2
  return p
```

Table 13: Combinators Implementation

the selective communication protocol found in Concurrent ML [15]. Details are available upon request.

Our implementation makes use of the STM-based core language. The helper go iterates over the list of promises. We abort (via `retry`) if one of the promises has already been completed. Otherwise, we accumulate a STM transaction `a1` to complete each promises. As the check that promises are incomplete and the action `a1` to complete them is executed atomically, we can guarantee the all or nothing property. Action `a2` accumulates the to be executed callbacks and will only be executed once atomic completion is successful. Implementations based on CAS and MVar are possible but we believe are less elegant and require more effort (e.g. establishing a locking order etc).

B FURTHER DERIVED FEATURES

Based on the more 'primitive' features we can derive some expressive combinators. Table 12 summarizes some combinators we find useful. Their implementation is given in Table 13. Many of these operations can be found in Scala FP and Twitter FP. Some operations such as `firstSucc` are not directly supported.

Table 14 shows code snippets for a holiday planning use case. We consider two alternative bookings to the US or Switzerland

```
book currency hotel pred = do
  f1 <- future $ currency
  f2 <- f1 `guard` (\rate -> return $ pred rate)
  f2 `followedBy` (\rate -> hotel rate)

booking = do
  f1 <- book (euroToUSD 50) hotelUS (>=60)
  f2 <- book (euroToCHF 50) hotelCH (>=60)
  f1 `orAlt` f2
```

Table 14: Holiday planning

where based on the exchange rate we book a hotel room. We give preference to the US booking request but could easily change the booking logic by for example using `firstSucc` instead to choose the first available successful booking.

To summarize. There is a huge design space of what further features shall be supported. Some can be derived from core features. Others may take advantage of specific core implementation details. In future work, we plan to carry out a more comprehensive study of useful features and applications that make use of them.

C SOUND PROGRAM TRANSFORMATION

We present a sound program transformation for tryCompleteWith but without moving any callbacks or reducing the number of intermediate promises. Our idea is that we maintain a set of links per promise. Every link refers to a target promises from a tryCompleteWith call. The set is initially empty and might get a new element per tryCompleteWith call. When a promise with a non-empty set is completed with a result value, it tries to complete all target promises from its set and executes their callbacks. This transformation does save the one additional callback per tryCompleteWith call. Besides, it allows us to collect all callbacks of the intermediate promises and execute them at once. However, it does not reduce the number of intermediate promises. The unsound example of section 6 which relied on rule (Try-W2b) can not be reproduced with this transformation which makes it sound.

We extend our language:

Definition C.1 (Core Syntax).

$$\begin{array}{lcll} l & ::= & p \mid q \mid r & \text{Link to promise} \\ ls & ::= & \{l_1, \ldots, l_n\} & \text{Set of links} \\ \delta & ::= & v \mid hs \mid \{hs, ls\} & \text{Promise State} \end{array}$$

The promsie state can now be a list of callbacks plus a set of links.

We change the rule (Try-W2b) into the rule (Try-W2c). This rule only adds a link to the target promise and does not move any callbacks.

Definition C.2.

$$\begin{array}{ll} \text{(Try-W2c)} & (\{p_{hs}, q_{gs}\} \cup \mathcal{P} \mid \text{tryCompleteWith } p\ q : cs) \\ & \Rightarrow \\ & (\{p_{hs}, q_{\{gs, \{p\}\}}\} \cup \mathcal{P} \mid \{\![[\text{tryCompleteWith } p\ q], cs]\!\}) \end{array}$$

Furthermore, we have to introduce two new rules (On-3) and (Try-3). Both rules are applied when the promise is not completed and has a set of links in its state. (On-3) simpliy adds the callback and keeps the set of links. (Try-3) will not only execute the callbacks with the result value but try to complete all links and collect their callbacks, too. After collecting all callbacks it will execute them at once in a single thread instead of executing each callback in a separate thread. If a linked promise is already completed, it will be ignored.

Definition C.3 (Updated Command Semantics).

$$\begin{array}{ll} \text{(On-3)} & (\{p_{\{hs, ls\}}\} \cup \mathcal{P} \mid \text{onComplete } p\ h : cs) \\ & \Rightarrow \\ & (\{p_{\{h:hs, ls\}}\} \cup \mathcal{P} \mid \{\!\{cs\}\!\}) \end{array}$$

$$\begin{array}{ll} \text{(Try-3)} & (\{p_{\{hs, ls\}}\} \cup \mathcal{P} \mid \text{tryComplete } p\ v : cs) \\ & \Rightarrow \\ (\{p_v\} \cup \mathcal{P} \mid \{\![[h_1(v), \ldots, h_n(v), tryCompletel_1 v, \ldots, tryCompletel_n v], cs]\!\}) & \end{array}$$

We have implemented the new rules in Scala for CAS only. We use the performance test shown in table 15 to compare the different rules. The test creates n promises and registers m callbacks per promise. It uses tryCompleteWith to complete each promise with the following one. The final promise is completed with an integer value and completes the whole chain of promises.

```
val n = 100
val m = 100000
var ex = getExecutor(executorThreads)
val promises = n times f(ex)

def linkPromises(promises : Seq[FP[Int]]) {
  m times promises(0).onComplete(_ => incCounter)
  if (promises.size == 1) {
    promises(0).trySuccess(1)
  } else {
    promises(0).tryCompleteWith(promises(1))
    linkPromises(promises.drop(1))
  }
}

linkPromises(promises)
await n * m
```

Table 15: Test 4: Chain onCompletes and tryCompleteWiths

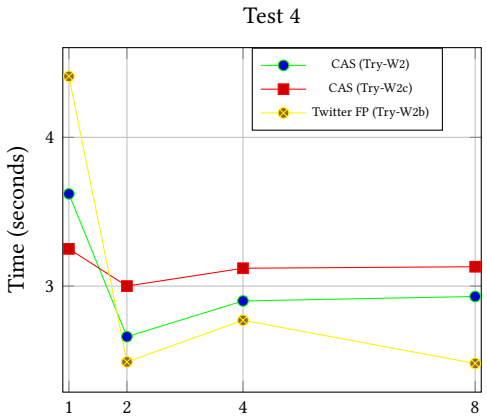

Figure 3: Benchmark for test 4 in Scala

We compare the performance of our previously shown CAS implementation ((Try-W2)) with a CAS implementation which uses the sound program transformation ((Try-W2c)) and Twitter FP ((Try-W2b)). We cannot compare with Scala FP because of the restrictions of the optimization. Figure 3 shows the results of the benchmark. For the standard CAS Implementation ((Try-W2)) it creates $2 * n * m$ callbacks. n callbacks come from tryCompleteWith per promise. For the other implementations ((Try-W2b) and (Try-W2c)) it creates $n * m$ callbacks. The callbacks of tryCompleteWith are removed due to the rule optimization. Besides, the CAS implementation with the sound program transformation ((Try-W2c)) creates only one single task for all callbacks instead of one task per callback. This improves the performance if there is only one executor thread. The reduced number of callbacks does not seem to improve the performance. Hence, our sound program transformation is only useful if the amount of concurrency should be decreased.