From Events to Futures and Promises and back

Martin Sulzmann

Faculty of Computer Science and Business Information Systems
Karlsruhe University of Applied Sciences
Moltkestrasse 30, 76133 Karlsruhe, Germany
`martin.sulzmann@hs-karlsruhe.de`

Abstract. Events based on channel communications and futures/promises are powerful but seemingly different concepts for concurrent programming. We show that one concept can be expressed in terms of the other with surprisingly little effort. Our results offer light-weight library based approaches to implement events and futures/promises. Empirical results show that our approach works well in practice.

1 Introduction

There exists a wealth of programming models to make concurrent programming easier. A popular form is Communicating Sequential Processes (CSP) [7] which has found its way into several programming languages in one form or another.

For example, Haskell [9] provides native support for a primitive channel type called MVar which is effectively a buffered channel of size one. The Go programming language [5] follows more closely CSP and has native support for synchronous channel-based message passing and selective communication. Concurrent ML (CML) [11] provides for even richer abstractions in the form of events.

Another popular concurrency approach is the concept of futures/promises [1, 4]. For example, Alice [12] provides native support for futures/promises whereas Scala [6] follows a library-based approach and provides a powerful set of combinators to compose futures/promises.

Our interest lies in establishing some connections among channel-based events and futures/promises by expressing one concept in terms of the other. This is interesting and helpful for several reasons. We can hope to obtain insights into commonalities and differences which are useful in choosing the most appropriate concept for a specific problem domain. As shown above, many languages support (parts of) one concept but lack the other. By being able to express one concept in terms of the other, we get one concept for free.

Earlier works There is some folklore knowledge how to express futures in terms of channels. This is indeed more of an exercise. For example, Reppy [10] shows how to express futures in CML based on the same approach we outline here. However, we are not aware of any works which like us cover promises and some powerful combinators. There is some, mostly superficial, discussions on internet forums [3, 8] related to the connection between events and futures. To the best of our knowledge, we are the first to describe an implementation of CML style events based on futures/promises. The only closely related work we are aware of is the work by Chaudhuri [2]. He shows how to express CML in terms of MVar. Our results provide for an alternative library-based implementation approach of CML and our experiments show that we are often faster.

Contributions and Outline In summary, we make the following contributions:

– First, we show how to express futures via CML style events (Section 3.1).
– Next, we observe that a simpler implementation, only relying on channels, is possible once we include promises (Section 3.2).
– Then, we show that based on futures/promises, CML style events can elegantly be implemented (Section 4).
– Finally, we report on our implementations in Go as well as in Haskell (Section 5).

The upcoming section provides background information on the concurrency models we study here. We will use Go syntax [5] throughout the paper. Our implementations in Go and Haskell can be accessed via [13].

2 Background

2.1 Channel-Based and Selective Communication

We consider languages supporting concurrent processes (a.k.a. threads), synchronous channel-based communication and selective communication. These are the key concepts found in Communicating Sequential Processes (CSP) [7]. Examples of programming languages providing support for these concepts are Concurrent ML (CML) [11] and the Go programming language [5].

Let us consider a simple example in Go illustrating these concepts. We start with the main function

```
func main() {
  x := make(chan string)
  y := make(chan string)
  go func() { x <- "hi" }()
  go func() { y <- "ho" }()
  v1, v2 := sel(x, y)
}
```

Functions are introduced via the `func` keyword. We create two channels over which we intend to transmit integer values. The type of channels `x` and `y` is implicit as Go supports a simple form of type inference. In case of assignment via `:=` the type of the left-hand side variable is inferred by the defining right-hand side.

Creation of threads [1] is done via the go keyword. The argument is a parameterless function with no return value. We make use here of anonymous functions introduced via func. In our example, both threads transmit some string to channels x and y where on the left of the arrow we find the channel and on the right the message. Message exchange is synchronous and follows the rendezvous principle. That is, a sender blocks unless there is a concurrent receiver over the same channel.

Let us consider the definition of sel.

```
func sel(x chan string, y chan string) (string,bool) {

  select {
  case v := <-x:   return v,true
  case v := <-y:   return v,false
  }
}
```

In Go, variable declarations first mention the variable name, e.g. ch1, and then its type, e.g. chan string. Furthermore, the return type appears after the list of formal parameters We make use here of a function which returns multiple values.

The interesting bit is statement select via which we can choose among multiple communications. Each case attempts to perform a receive operation where we place the channel to the right of the arrow. If available the received message is bound to some local variable. If none of the cases applies, the select statement will block. If both cases are available, one of the cases will be chosen where the choice is non-deterministic. [2]

2.2 Events for First-Class Communication Patterns

The CML language goes one step further compared to Go by providing a powerful, modular style of programming in the form of events. Events are an abstraction to describe complex communication patterns. They are built using primitive send/receive operations as well as selective communications and can be passed around as values.

Here is a recast of CML style event combinators in terms of Go syntax.

```
type Channel<T>
type Event<T>
func syncEvt(Event<T>) T
func newChannel() Channel<T>
```

[1] In Go, threads are referred to as go-routines. These are light-weight threads and scheduled cooperatively. The specific scheduling strategy does not matter for the purpose of this paper.

[2] The actual Go run-time implementation performs a pseudo-random permutation of all cases before trying out each case in a specific order. The first available of the such permuted cases will be chosen. Hence, the order of the cases in a select statement does not matter.

```
func sndEvt(Channel<T>,T) Event<()>
func rcvEvt(Channel<T>) Event<T>
func chooseEvt(Event<T>, Event<T>) Event<T>
func (Event<T>) wrap(func(T) S) Event<S>
func spawn(func())
```

For convenience, we pretend that Go supports generics, i.e. parameteric polymorphism, using Java syntax. We also assume a unit type () with value () as the only inhabitant. We further assume that Go has first-class support for pairs and not only for multiple return values. Thus, we can remove some 'noise' from our example programs.

We explain the various event combinators by recasting the above example.

```
func selCML(e1 Event<(string,bool)>,
            e2 Event<(string,bool)>) (string,bool) {
     return syncEvt(chooseEvt(e1, e2))
}

func main() {
  x := newChannel()
  y := newChannel()
  spawn func() { syncEvt(sndEvt(x, "hi")) }()
  spawn func() { syncEvt(sndEvt(y, "ho")) }()
  selCML(rcvEvt(x).wrap(func(v string) { return v,true }),
         rcvEvt(y).wrap(func(v string) { return v,false }))
}
```

As before we create two channels and two threads. Unlike in (plain) Go, primitives sndEvt and rcvEvt for transmission and reception yield an event. Events are not 'executed' until we attempt to synchronize them via syncEvt. Hence, we can pass around events as values.

Consider the arguments in the call to selCML, for example

```
rcvEvt(x).wrap(func (v string) { return v,true }
```

The above event describes reception of a value over channel `x`. On the resulting value, we perform a post-processing action via `wrap`. In Go syntax, `wrap` is a method where the event plays the role of an object. The actual argument of `wrap` is the post-processing function. Function selCML then simply chooses among one of the two events. In `main` we ignore the return values of selCML for brevity.

This style of event-based programming leads to highly modular and reusable code. As said, Go only supports plain channel-based and selective communications without any further abstractions. Later we will show how to implement such a CML style event combinator library based on plain channels only.

2.3 Futures/Promises for Composable Asynchronous Computations

Futures [1] are yet another powerful concurrency model. A future serves as a place-holder for a read-only variable whose result is as yet undetermined but will be eventually become available. From the user point of view, programming with futures feels close to the common 'sequential' programming model. The underlying implementation carries out computation steps asynchronously as much as possible by relieving the user from the burden of explicit thread management.

Like in case of CML, modern languages like Scala support a rich set of combinators [6]. Here are some of these combinators in terms of Go syntax.

```
type Future<T>
func future(func () (T,bool)) Future<T>
func get(Future<T>) (T,bool)
func onSuccess(Future<T>, func (T))
func onFailure(Future<T>, func ())
func alt(Future<T>, Future<T>) Future<T>
func (Future<T>) then(func(T) (S,bool)) Future<S>
```

Function `future` takes a function which will be executed asynchronously (in the background). This (parameterless) function either yields some result or fails due to some exceptional behavior. We follow here Go's philosophy that exceptional behavior shall be indicated by an additional (Boolean) return value. In practice, we might want to choose a more informative type than bool. We use bool for brevity. Hence, the return type (T, bool).

We can query the result of this computation via the (blocking) get function. Alternatively, we can use the non-blocking versions onSuccess and onFailure which apply some callback functions once the (future) computation result becomes available. Function `alt` is a combinator to (randomly) select the first available future. Function `then` allows for some post-processing of (future) computations.

A closely connected concept are promises. While a future are read-only and their result can be queried many times (once the result becomes available), a promise represents a write-once data container which is initially empty. Each promise implies a future and the promise can be completed at a specific program point. Here are the essentials of promises.

```
type Promise<T>
func (Promise<T>) future() Future<T>
func trySuccess(p Promise<T>, v T) bool
func tryFail(p Promise<T>) bool
```

Method `future` is applied on a promise object and yields the future underlying the promise. Via `trySuccess` we can either successfully complete or fail a promise. A promise can be completed/failed only once. Hence, the Boolean return value to indicate if the 'try' was successful.

Here is a recast of our running example in terms of futures and promises.

```
func main() {
  x := promise()
  y := promise()
  f := alt(x.future().then(func(v string) (string,bool)
                               { return v,true }),
               y.future().then(func(v string) (string, bool)
                               { return v,false })
  trySuccess(x, "hi")
  trySuccess(y, "ho")
  get(f)
}
```

We wish to transmit two messages. Our only means for transmission is to introduce two promises. Selection among the two messages is done via future `f`. From each promise we derive the underlying future on which we apply some post-processing function. Completion of both promises can take place later. The blocking call get then finally retrieves the selected result.

2.4 Events versus Futures/Promises

Both concepts appear to be closely related, yet, there are some significant differences. Events as well as futures provide combinators to build richer programming abstractions. For example, for both concepts we find combinators for selection and post-processing. On the other hand, futures can be read many times but promises can be only written only whereas channels can be read and written arbitrarily often. Furthermore, selection in case of events (chooseEvt) is exclusive. That is, only the chosen event will actually take place. In case of futures (alt), both futures may succeed but only one of the results will be chosen. Albeit these differences, we will show that one can be expressed in terms of the other.

3 From Events and Channels to Futures/Promises

We first show how to express futures in terms of CML style event combinators. As there are a number of languages which support channels but lack further abstractions such as selective communication etc, we also provide for an implementation of futures and promises which only requires plain channels.

3.1 Event-Based Futures

A future is represented by an event which either yields a value or reports failure. Thus, the blocking get function simply needs to synchronize the (future) event.

```
type Future<T> = Event<(T,bool)>

get(f Future<T>) (T,bool) {
   return syncEvt(f)
 }
```

The actual 'future' computation is executed in its own thread and the result is retrieved via a channel.

```
func future(func f() (T,bool)) Future<T> {
  ch = newChannel()
  spawn(func() {
          x := f()
          for { syncEvt(sndEvt(ch, x)) }
       })
  return rcvEvt(ch)
}
```

For convenience, we repeatedly transmit the computed value over the channel. Thus, we support the read-many times semantics. We could avoid the infinite loop and have the get function re-submit the (future) value. However, then we would need to expose the channel underlying the future. Regardless which option we favor, the remaining combinators of the future library are easily implemented.

To query the result in a non-blocking fashion, we simply create a thread within which we call the blocking get.

```
func onSuccess(f Future<T>, func cb(T)) {
  spawn(func() {
         v,s := get(f)
         if s { cb(v) }
       })
}

func onFailure(f Future<T>, func cb()) {
  spawn(func() {
         v,s := get(f)
         if !s { cb() }
       })
}
```

Selection among futures can be straightforwardly mapped to chooseEvt. Post-processing is by retrieving the future value via code in a newly generated future.

```
func alt(f1 Future<T>, f2 Future<T>) Future<T> {
    return chooseEvt(f1,f2)
}

func (f Future<T>) then(w func(T) (S,bool)) Future<S> {
  return future(func() (S,bool) {
                  v, o := f.get()
                  if o {  return w(v) }
                  return nil, false
         })
}
```

The implementation of alt suggests that the full power of CML style events is required. As we show in the following, plain channels are sufficient.

3.2 Channel-Based Futures and Promises

We assume synchronous channels as found in CML where we write receive(x) as a short-hand for syncEvt(rcvEvt(x)) and transmit(x,v) as a short-hand for syncEvt(sndEvt(x,v)).

A future is now represented as a synchronous channel in which we place the value once available. The channel-based get primitive is similar to its event-based variant. As the channel is available, we assume it is the duty of get to make the computed value available for further reads.

```
type Future<T> = Channel<(T,bool)>

  func future(func f() (T,bool)) Future<T> {
  ch = newChannel()
  spawn(func() {
            x = f()
            transmit(ch, x)
         })
  return ch
}

get(f Future<T>) (T,bool) {
   x = receive(f)
   spawn(func() { transmit(f,x) })
   return x
}
```

The definitions of functions onSuccess, onFailure and then remain unchanged. We ignore alt for the moment and consider promises next.

A promise is represented by a future (also a channel) and another channel which holds the state of the promise. The state of the promise is either full or empty and represented by a Boolean value. The state is recorded in a channel to guarantee exclusive access.

```
type Promise<T> = (Future<T>, Channel<bool>)

func (p Promise<T>) future Future<T> {
  return fst(p)
}

func promise() Promise<T> {
  f = newChannel()
  e = newChannel()
  spawn(func() { transmit(e,true) })
  return (f,e)
}
```

Extraction of the promise is by projection onto the first component which is done via primitive fst. Initially, we set the status of the promise to empty by transmitting (true) to the channel. As we assume synchronous channels, the transmission must take place in its own thread.

Definitions to either successfully complete or a fail a promise are as follows.

```
func trySuccess(p Promise<T>, v T) bool {
  b := receive(snd(p))
  if b { spawn(func() { transmit(fst(p), (v,true)) }) }
  spawn(func() { transmit(snd(p), false) })
  return b
}

func tryFail(p Promise<T>, v T) bool {
  b := receive(snd(p))
  if b { spawn(func() { transmit(fst(p), (nil,false)) }) }
  spawn(func() { transmit(snd(p), false) })
  return b
}
```

We first query the state of the promise via b := receive(snd(p)) which also locks the promise. If still empty, we succeed or fail the promise by transmitting the value to the channel which represents the future. We assume that nil represents a default value of any type. We then unlock the promise and report if completion was successful.

Finally, let us consider the definition of alt. The trick is to define a function via which we use a future to complete a promise. Then, we start a race among two futures trying to complete a promise. Here are the details.

```
func tryCompleteWith(p Promise<T>, f Future<T>) {
  spawn(func() {
            v,s = get(f)
            if s { trySuccess(p,v) }
            else { tryFail(p) }
         })
}

func alt(f1 Future<T>, f2 Future<T>) Future<T> {
  p = promise()
  tryCompleteWith(p, f1)
  tryCompleteWith(p, f2)
  return p.future()
}
```

4 Events based on Futures/Promises

Based on the earlier observations in Section 2.4, it is clear that we must assume channels for primitive communications. In addition, we require a modest extension of the completion mechanism for promises. But that is all we need to obtain a working solution within a few lines of code.

4.1 Events are Futures which share a Promise

An Event is effectively represented by a future. One thing we need to take care of is that in case there is a choice among several events, only one of them is allowed to succeed. For example, consider chooseEvt(sndEvt(ch, 1), sndEvt(ch, 2)). There are two send operations competing with each other. Hence, we need some mechanism which guarantees that only one of these (future) events will succeed.

Our idea is to supply each alternative with a promise which is common to all alternatives. We start a race among all alternatives where each alternative tries to succeed this promise. Based on the 'write-once' semantics of promises there can be at most one winner. Hence, a event is represented as a function from a promise to a future.

```
type Event<T> = func(Promise<T>) Future<T>
```

To synchronize an event, we simply create a promise, call the event with that promise and then retrieve the (future) value via get.

```
func syncEvt(e Event<T>) T {
  env = promise()
  v, _ = get(e(env))
  return v
}
```

For exclusive selection among two events, we simply retrieve the futures underlying both choices by application to the common promise. At the level of futures, we then perform the selection via alt. The upcoming rendezvous protocol guarantees that only ever one of both futures can be successful.

```
func chooseEvt(e1 Event<T>, e2 Event<T>) Event<T> {
  return func(env Promise<T>) Future<T> {
                f1 = e1(env)
                f2 = e2(env)
                return alt(f1,f2)
         }
}
```

Functions for post-processing and spawning of threads are straightforward.

```
func (e Event<T>) wrap(w func(T) S) Event<S> {
  return func(env Promise<S>) {
          f = e(env)
```

```
              p = promise()
              onSuccess(f, func(v S) { trySuccess(p, w(v)) })
              return future(p)
            }
}

func spawn(func f()) {
  future(func() (int, bool) {
              f()
              return nil,true
            })
}
```

4.2 Rendezvous via Atomic Completion

What remains is to consider synchronization among primitive events which follow the rendezvous principle. The task is to find matching rendezvous partners which communicate over a common channel. We assume that transmit plays the passive role whereas receive plays the active role. Transmit passes the value and its environment promise to receive. Receive starts a race and tries to succeed its own environment promise and the environment promise of its rendezvous partner. As there are other receives searching concurrently for rendezvous partners, this step needs to be done *atomically*. That is, all or nothing.

Race among Rendezvous Partners For example, consider synchronization of the following three concurrent events

$$\text{sync}(\text{chooseEvt}(\underbrace{\text{receive}(\text{ch1})}_{r_1}, \underbrace{\text{receive}(\text{ch2})}_{r_2}))$$

$$\text{sync}(\underbrace{\text{transmit}(\text{ch1,1})}_{s_1}) \qquad \text{sync}(\underbrace{\text{transmit}(\text{ch2,2})}_{s_2})$$

Receives r_1 and r_2 are independent and start a race to find matching rendezvous partners. Suppose r_1 is paired up with s_1 and r_2 is paired up with s_2. Each pair is represented by their environment promises. A winning rendezvous pair is found if both their environment promises can be successfully completed.

In our example, r_1 and r_2 stem from the same chooseEvt statement and therefore share the environment promise. Hence, each receive must try atomically to complete its environment promise and the promise of the rendezvous partner. This guarantees that either r_1 synchronizes with s_1 or r_2 synchronizes with s_2.

Atomic Completion The extension to guarantee atomic completion of two promises is fairly straightforward. We assume that channels are comparable. Thus, we establish a global locking order to atomically access two promises.

```
func trySucc2(p1 Promise<T>, v1 T,
              p2 Promise<S>, v2 S) bool {
  if snd(p1) < snd(p2) {
    return trySucc2LO(p1,v1,p2,v2)
  }
  return trySucc2LO(p2,v2,p1,v1)

}
```

Based on the locking order, we can guarantee that helper function `trySucc2LO` will not deadlock. We check if both promises are still empty. If yes, we found a winner and complete both. Otherwise, we abort.

```
func trySucc2LO(p1 Promise<T>, v1 T,
                p2 Promise<S>, v2 S) bool {
  b1 = receive(snd(p1))
  b2 = receive(snd(p2))
  if b1 && b2 {
    spawn(func() { transmit(fst(p1), (v1,true)) })
    spawn(func() { transmit(fst(p2), (v2,true)) })
    b1 = false
    b2 = false
  }
  spawn(func() { transmit(snd(p1), b1) })
  spawn(func() { transmit(snd(p2), b2) })
  return !b1 && !b2
}
```

Retry for Losers In case of abort, certain follow-up actions may be necessary. Recall the above example. Suppose the winner is r_1 and s_1. Hence, receive r_2 has lost but still holds the information transmitted by s_2. It is entirely possible that s_2 could still be synchronized. For example, suppose we encounter a fourth concurrent event of the form

$$\text{sync}(\underbrace{\text{receive(ch2)}}_{r_3})$$

Therefore, the loser r_2 must re-transmit the information received from s_2 such that r_3 is able to synchronize with s_2.

It is also possible that r_2 drops s_2 but still has a chance to find a match. As another example, consider

$$\text{sync}(\text{chooseEvt}(\underbrace{\text{transmit(ch1,1)}}_{s_1}, \underbrace{\text{transmit(ch2,2)}}_{s_2}))$$
$$\text{sync}(\underbrace{\text{receive(ch1)}}_{r_1}) \qquad \text{sync}(\underbrace{\text{receive(ch2)}}_{r_2})$$

Suppose that r_1 synchronizes with s_1. Receive r_2 holds information obtained from s_2. As s_2 has clearly lost, there is no point for r_2 to re-transmit s_2's information. However, it is entirely possible that r_2 can still be synchronized. For example, consider

$$\text{sync}(\underbrace{\text{transmit(ch2,3)}}_{s_3})$$

Promise for Post-Processing So far, we assumed that a channel value only needs to consist of the to be transmitted value and the environment promise to decide the winning pair of rendezvous partners. That is not quite sufficient yet.

For example, suppose the following event can be synchronized

sync(receiveEvt(ch).wrap(...), receiveEvt(ch).wrap(...))

We must ensure that only the 'wrap' action of the event involved in the synchronization (either the first or second) is executed. As the environment promise is common to both events, we require a promise for each rendezvous partner. If successful, we propagate this information. See `wrap`.

Rendezvous Implementation Let us take a look at the details.

```
type Channel<T> = ChannelN<(T, Promise<()>, Promise<()>)>

func newChanenl() ChannelN<T> {
  return newChannelN()
}

func sndEvt(ch Channel<T>, x T) Event<()> {
   return func(pEnv Promise<T>) Future<T> {
          pSnd := promise()
          spawn(func() { transmitN(ch,(x, pEnv, pSnd)) })
          return future(pSnd)
          }
}
```

Native channel operations carry the suffix 'N' to avoid confusion with the to be defined channel and event abstractions. A channel is based on a native channel consisting of the to be transmitted value, the environment promise and a promise to identify each rendezvous partner. As said, sndEvt plays a passive role and only transmit its information so that some rendezvous partner can pick up this information. The rendezvous partner will set promise pSnd if this event has been involved in a successful synchronization step.

Finally, let us consider rcvEVt which plays an active part in the search for a matching rendezvous partner.

```
func rcvEvt(ch Channel<T>) Event<T> {
  return func(pEnv Promise<T>) Future<T> {
       pRcv = promise()
       spawn(func() {
          for {
            rv     := receiveN(ch)                         // P0
            v      := fst(rv)
            rvEnv  := snd(rv)
            rvSnd  := third(rv)
            if pEnv == rvEnv {
              spawn(func() { transmitN(ch, rv) })    // P1
            } else {
              if trySucc2(pEnv, (), rvEnv, ()) {     // P2
                    trySuccess(rvSnd, ())
                    trySuccess(pRcv, v)                // P3
                    goto STOP
              }
              if isEmpty(rvEnv) {                        // P4
                    fork(func() { transmitN(ch, rv) })
                    goto STOP
              }
              if !isEmpty(pEnv) {                        // P5
                    goto STOP
              }
            }
          } // for
          STOP:
       })
       return future(pRcv)
      }
 }
```

We look for a potential matching partner. See program point P0. It is entirely possible that we pick a partner from the same environment. For example, consider

$$\text{sync}(\text{chooseEvt}(\underbrace{\text{rcvEvt(ch).wrap(...)}}_{r_1}, \underbrace{\text{sndEvt(ch,1)}}_{s_1}))$$

Then, we must put back this choice. See program point P1. If this is the only event available, we end up in a busy loop where r_1 keeps selecting and aborting s_1 over and over again. This seems wasteful but as this is clearly a corner case there is no practical issue.

Suppose, we encounter another concurrent event

$$\text{sync}(\text{chooseEvt}(\underbrace{\text{rcvEvt(ch).wrap(...)}}_{r_2}, \underbrace{\text{sndEvt(ch,2)}}_{s_2}))$$

There might be the danger that r_1 repeatedly picks s_1 and r_2 picks s_2. However, we assume that receiving and sending parties on a common channel are queued.

Hence, there is the guarantee that any of the potential matches, either (r_1, s_1) or (r_2, s_2), will be encountered eventually.

If atomic completion is successful, we propagate that information to the rendezvous partners involved. See program point `P2`. Note that the transmission of the actual value takes place via a promise. See program point `P3`. Otherwise, there are two possible follow-up actions (recall above discussion *Retry for Losers*).

We assume a primitive `isEmpty` to query the state of a promise. Such a primitive is easy to implement and omitted for brevity. If our rendezvous partner is still in the game to be matched, see program point `P4`, we put back this choice and exit as the receive must be the reason for failure. We also stop if the receive is no longer in the game to find a matching partner. See program point `P5`.

5 Implementation

We have fully implemented the approach in Go as well as in Haskell. Both languages lack native support for futures and promises. Hence, we only evaluate the performance of CML style events as described in Section 4.

5.1 Implementation and Experiments in Go

When compared to native Go channels and selective communications, our Go-CML variant can be up to 100 times slower. This is no surprise as we compare a highly-tuned native CSP implementation against a library-based approach which supports powerful event combinators. Our implementation consists of less than 300 lines of Go code whereas the native Go implementation is much more involved.

5.2 Implementation and Experiments in Haskell

Haskell also lacks native support for futures and promises but supports lightweight threads and MVar which is a one place channel. Section 3.1 assumes synchronous channels but with some minor modification an MVar-based implementation can be derived. The same applies to our encoding of events in terms of futures and promises.

Our Haskell implementation is very close to the code samples given in the paper. Slightly more concise and thanks to Haskell's support for parametric polymorphism, we can provide stronger static guarantees compared to the Go version. The channel-based futures and promises part comprises of about 150 lines of Haskell code, the CML part of about 100 lines of Haskell. But this also includes some commentary. In addition, we also support `wrapAbort` and a guard primitive as found in CML.

We compare ourselves against [2] which like us considers a library-based approach to encode CML in Haskell and also relies on MVar. However, the approach is rather different as we mainly base our approach on futures and promises to encode the non-trivial synchronization patterns in CML. Overall, the implementation in [2] consists of about 150 lines of code. So in terms of code size both approaches are comparable.

In our experiments we use the benchmarks described in [2] as well as our own. Details concerning the implementation and our benchmark examples can be found here [13]. In general, we can report that our implementation is slightly faster (10-30%). We encountered one (artificial) example where our implementation is slower (about 20%). The examples make use of many channels where there is a large choice among events (deeply nested chooseEvt).

6 Conclusion

Our main contribution is a novel library-based approach to implement CML style event synchronization patterns based on futures and promises. We provide implementations in Go and in Haskell. These language have no native support for future and promises. Hence, we show how to easily implement them based on plain channels. Our experiments in Haskell show that our approach is competitive in comparison to earlier works. In case of Go, we can provide a much richer set of event abstractions than is currently available.

References

1. Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.
2. Avik Chaudhuri. A concurrent ML library in concurrent Haskell. In *Proc. of ICFP'09*, pages 269–280, New York, NY, USA, 2009. ACM.
3. C++ futures. http://lambda-the-ultimate.org/node/3221, 2009.
4. Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing, 1976.
5. The go programming language. https://golang.org/.
6. Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. http://docs.scala-lang.org/overviews/core/futures.html.
7. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
8. Bartosz Milewski. Futures done right. http://bartoszmilewski.com/2009/03/10/futures-done-right/, 2009.
9. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proc. of POPL'96*, pages 295–308, New York, NY, USA, 1996. ACM.
10. John H. Reppy. *Higher-order Concurrency.* PhD thesis, Cornell University, 1992. Available as Computer Science Technical Report 92-1285.
11. John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, New York, NY, USA, 1999.
12. The Alice Manual. http://www.ps.uni-sb.de/alice/manual/.
13. Martin Sulzmann. From events to futures and promises and back - links to implementations and benchmark examples in Go and Haskell. `http://www.home.hs-karlsruhe.de/~suma0002/events-futures.html`.