Practical Abstractions for Concurrent Interactive Programs

Stefan K. Muller **William A. Duff**
Umut A. Acar

August 2015
CMU-CS-15-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is an expanded version of a paper that was submitted to the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2015) in November 2014. Apart from the inclusion of additional technical details, the original submission has only been modified to make typographical corrections.

This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, by the European Research Council under grant number ERC-2012-StG-308246, and by Microsoft Research.

Keywords: interaction, reactive programming, performance evaluation, programming language design

Abstract

We propose abstractions for designing and implementing concurrent interactive applications, such as graphical user interfaces, games and simulations, that interact with the external world in complex ways. The abstractions enable the programmer to write software that interacts with external agents (e.g., mouse, keyboard, network, the operating system) both synchronously and asynchronously in a uniform manner by means of several simple (concurrent) primitives. We specify the semantics of the primitives and provide an implementation as an OCaml library. We develop a software framework for assessing the responsiveness of interactive applications and evaluate our techniques by considering a range of applications including games, physics simulations, a music-streaming server, and a Unix shell, as well as microbenchmarks that enable quantitative analysis. Our results show the proposed abstractions to be expressive and performant.

1 Introduction

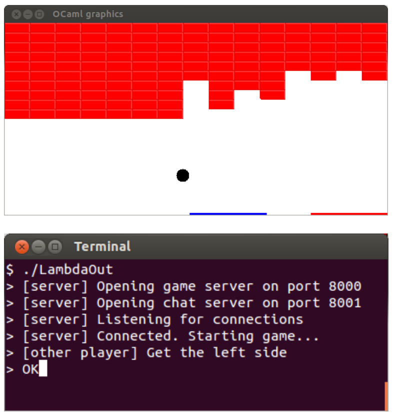
Some of the most interesting and complex software involves interaction with an external agent such as a user, another software system, or a device such as a sensor. Such programs, called interactive or sometimes reactive, can be challenging to design, implement, and reason about because they are usually designed to run forever, and because their semantics and runtime behavior depend on their interaction with the environment.

As an example, consider a multiplayer online arcade-style game (inspired by *"Breakout"*) that allows two players connected via a network to collaborate in knocking out several rows of bricks displayed on the screen; Figure 1 shows a snapshot from our mock-up implementation of such a game. To play the game, each player starts the game on his or her computer, which displays a graphics window and a chat window and updates them based on the input from both users by communicating via the network as necessary. The users can communicate via the chat window, for example to determine a strategy.

To develop an efficient and responsive interactive system, such as the multiplayer game, the programmer needs a programming language and system that guarantee the following three *effectiveness criteria*:

- **Expressiveness.** Interactive applications can interact with many sources of input and also perform complex computation. For example, a realistic multiplayer game may interact with multiple users via mouse, keyboard, and network and manipulate 3D objects. The language therefore must be as general-purpose as possible.
- **Control over sampling/polling.** Interactive systems must sample (poll) input sources and update output sources at periodic intervals. For correctness and responsiveness, it is critical for the programmer to be able to control the frequency of sampling. In the multiplayer game example, each input source has its own natural sampling rate: game-control keys should be sampled more frequently than the network connection used for infrequent chat messages. It may also be important to change the frequency of sampling dynamically. For example, when objects move quickly, the calculations should be updated more frequently to avoid missing collisions.
- **Concurrency.** Reducing sampling frequency can reduce unnecessary work but only up to a limit, because it increases response time and can decrease accuracy of calculations. It should therefore be possible to sample asynchronously, i.e., be notified when data becomes available and compute a response concurrently. In the multiplayer game, it is more responsive and more efficient to receive the occasional chat message asynchronously, rather than periodically sampling the network.

Interactive programs can be written using *event-driven programming*, where an event loop waits for events and handles them by scheduling for execution the appropriate *callback functions* or *event handlers*. Event-driven programming meets the effectiveness criteria described above but unfortunately, for well-documented reasons (e.g., [12, 16, 9]), writing event-driven programs is notoriously difficult. Perhaps the most important difficulties stem from the fact that event-driven programs break key programming abstractions such as the function-call abstraction. For example,



```
$ ./LambdaOut
> [server] Opening game server on port 8000
> [server] Opening chat server on port 8001
> [server] Listening for connections
> [server] Connected. Starting game...
> [other player] Get the left side
> OK
```

Figure 1: Screenshots of a simplified multiplayer game: the main window (top) and the terminal chat window (bottom).

callbacks don't always return to their caller and they can be scheduled by interactive events or by other callbacks. In addition, these programs rely on side-effects for communication between callbacks. Due to this complexity, event-driven programs are sometimes described by colorful terms such as "callback hell" [12].

To raise the level of abstraction for writing interactive programs, prior research proposed Functional Reactive Programming (FRP) [13]. While theoretically appealing, FRP proved to be challenging to implement. Based on the work of Wan and Hudak [41], existing implementations of FRP adopt a synchronous evaluation strategy [19, 6, 4], which limits control over sampling frequency of interactive inputs [35, 11]. Such implementations can also suffer from difficult performance problems called time and space leaks [35, 31, 30]. Finally, although there has been recent interest in adding some concurrency support to FRP with the Elm language [1, 11], the implementation of Elm supports very limited forms of concurrency. These limitations can lead functional reactive programs to fail to perform as expected. For example, in Section 4, we empirically show how a functional reactive program (written in Elm) can fail to count the number of mouse clicks responsively.

Thus, on the one hand, we have event-driven programming, which satisfies the aforementioned effectiveness criteria but requires writing programs at a low level of abstraction. On the other hand, we have Functional Reactive Programming, which offers a very high level of abstraction but leads to limited expressiveness, limited control over sampling, and limited or no support for concurrency. In addition, empirical work on the performance and efficiency properties of interactivity abstractions is quite scarce.

In this paper, we present high-level linguistic abstractions for interaction that meet the effectiveness criteria as well as empirical techniques for evaluating the effectiveness of interactive programs. Our approach revolves around the following goal: develop primitives that encapsulate the essence of interaction—i.e., information exchange—in a single data structure and make it possible for this data structure to be used as a first-class value in a higher-order language. The data structure should be powerful enough to support implicit concurrency, similar to implicit parallelism techniques such as fork-join parallelism and futures [20, 15, 18, 27, 5, 24, 17, 28, 8] that enable expressing concurrency at a high level, dramatically simplifying developing concurrent programs.

At a high level, our goal is to make interaction just an ordinary operation, rather than a special form of computation.

To achieve this goal, we propose a first-class data structure which we call a *factor*, which can be dynamically created and can be queried both synchronously and asynchronously. A synchronous query supplies the factor with a value, called a *prompt*, and waits for the factor to respond, returning the response as the result of the query. An asynchronous query supplies the prompt to the factor but may not always return a response: it returns a response if one is available immediately, or starts a concurrent computation for computing the response and returns a *future factor* that can be queried later to obtain the response (synchronously or asynchronously).

Based on the idea of factors, we present a set of primitives that can be used to extend an existing higher-order language such as ML and specify the semantics of these primitives using Concurrent ML-style pseudo-code. We then implement these primitives as an OCaml library and evaluate their effectiveness by considering both expressiveness and efficiency. The library enables the expression of interactive programs within the OCaml language in a natural and structured style, allowing full use of OCaml features such as references and the foreign function interface.

To evaluate the expressiveness of the proposed techniques, we developed a reasonably broad range of applications ranging from games to a Unix shell, which involve, for example, complex interactions with the user and operating system. In addition, we used our library to implement the abstraction of *futures*, as well as previously proposed FRP techniques, namely Arrowized FRP [35] and Elm [11], as well as several accompanying examples. In fact, for Elm, we were able to implement the original concurrent model, which the existing Elm implementation does not support.

Evaluating the responsiveness of our techniques was much less straightforward: there is relatively little prior work on empirical studies of interactivity abstractions, primarily because of the technical difficulties of doing so (discussed in more detail in Section 4). For example, since its introduction in 1997, researchers have implemented a handful of different FRP systems [42, 43, 35, 31, 30, 26, 11], but, to the best of our knowledge, there are no empirical comparative studies of these systems. We developed a software framework for generating interactive inputs and evaluated the responsiveness of our techniques by developing several benchmarks and comparing responsiveness with respect to event-driven implementations. Our measurements show that the proposed techniques perform competitively with event-driven programming.

2 Language Design and Semantics

We present the interface for the abstractions that we propose. We describe their semantics informally, then algorithmically, and briefly describe how they are formalized. Throughout the presentation, we use simple examples, many of which are drawn from our implementation of the multiplayer game example of Section 1. To specify the interface, the semantics and the examples, we use OCaml-like syntax. For simplicity in presentation, we allow a form of *overloading* by using an operation at different types. OCaml does not support this form of overloading, but we are able to implement this behavior in our library using polymorphism. We use the term $\boldsymbol{\lambda}^{\mathbf{i}}$ ("lambda-i", where "i" stands for "interactive") to refer to a functional language extended with the proposed primitives.

```
module type Interactive = sig
  type ('p,'r) ftr
  type ('p,'r) fftr
  type ('p,'r) aview = Now of 'r * ('p,'r) ftr
                     | Later of ('p,'r) fftr

  (* Create a new factor from a generator. *)
  val ftr: ('p → 'r * ('p,'r) ftr) → ('p,'r) ftr

  (* Synchronously query a factor. *)
  val query: ('p,'r) ftr → 'p → 'r * ('p,'r) ftr
  val query: ('p,'r) fftr → unit → 'r * ('p,'r) ftr

  (* Asynchronously query a factor. *)
  val aquery: ('p,'r) ftr → 'p → ('p,'r) aview
  val aquery: ('p,'r) fftr → unit → ('p,'r) aview

  (* Split an I/O factor into two. *)
  val split : ('p,'r) ftr → ('p,'r) ftr * ('p,'r) ftr
end
```

Figure 2: The $\boldsymbol{\lambda}^{\mathbf{i}}$ core interface

2.1 The Primitives

Figure 2 shows the core of the interface for our interaction library, in a style similar to an ML module definition.

Factors. Our approach revolves around an interactivity abstraction that we call a *factor*. The type of a factor, `('p, 'r) ftr`, is parametrized by two types, the prompt type `'p` and the response type `'r`. Factors are first-class values, and thus can, for example, be created dynamically, passed to functions as arguments, and stored in other data structures. The primary operation over factors is *querying*. A factor is queried with a *prompt* of type `'p` and returns a *response* (of type `'r`) and a new factor, the *continuation.*

The prompt-response model for querying abstracts interaction as a two-way information exchange. Since, when queried, a factor returns a continuation factor, which enables future interaction, factors can effectively maintain internal state, passing information from now into the future. The primitive `split` simply creates a duplicate handle—an alias—for a factor, allowing it to be used multiple times.

To represent concurrently running computations, we define *future factors*, which have the type `('p, 'r) fftr`, where, as with factors, the type is parametrized by prompt and response. As we will see, future factors can only be created as a result of asynchronous queries. As concurrently running computations, future factors can only be prompted by a unit value when querying and cannot be split.

We rely on a linear type system (Section 2.3) to ensure that a queried (future) factor is never used again after it is queried, requiring instead the use of the continuation returned by the query. When paired with `split`, which creates a new alias for a factor, the linear type system causes no loss of generality but enables reasoning about interactive programs at a high level of abstraction, preserving, for example, referential transparency. Since it is restricted only to factors, linear use of factors is quite lightweight, leading to no noticeable cost in programming. In fact, it can be viewed as nothing more than witnessing/marking the use and sharing of factors.

Generators and defining factors. Programmers can create a factor by supplying a *generator* to the function `ftr`. A generator is a function that takes a prompt and produces a response and a continuation. Since factors can be defined rather liberally from any generator function, they can represent many different kinds of interactive computations. We distinguish between *I/O factors* that directly interact with the external world by performing I/O (through a system call or other effectful behavior) and *internal factors* or simply *factors* that that do not perform I/O directly but can indirectly perform I/O by querying an I/O factor.

Example 1 (I/O Factors: `std_in, network_resp`). An implementation of $\boldsymbol{\lambda}^{\mathbf{i}}$ will typically provide a standard library that defines I/O factors, allowing the programmers to work only with internal factors. However, for the purpose of illustration, we show here the definitions of some I/O factors used in the multiplayer game.

The I/O factor `std_in`, when prompted with a unit value, returns a line from standard input. It is defined using a generator that calls the OCaml function `read_line` and returns its result along with a continuation created by a recursive use of the generator.

```
let std_in : (unit, string) ftr =
  let rec std_in_gen () = (read_line (), ftr std_in_gen)
  in ftr std_in_gen
```

We can similarly define `network_resp`, a function which, when given a network socket, produces an I/O factor that, when queried with a prompt consisting of a string msg and an integer n, sends msg over the network and listens for n bytes in response, which it returns.

```
let rec network_resp sock =
  ftr (fun (msg, n) →
        ignore (sock_send sock msg);
        (sock_recv sock n, network_resp sock))
```

Querying factors. The synchronous `query` function queries a factor with a given prompt, waits for the response and continuation, and returns them when they become available.

Example 2 (Synchronous query). In our multiplayer game, the main loop uses the I/O factor `network_resp` to inform the other player of this player's actions. To this end, it sends a byte to the other player indicating the key currently pressed and waits for a byte in response. Since it is important for both players to operate in synchrony, we use `query` (rather than the asynchronous query discussed later).

```
let (my_key, key_presses') = (* get currently pressed key *) in
let (other_key, network_resp') =
  query network_resp (String.make 1 my_key, 1)
in
...
```

Example 3 (Internal factor: `map`). As first-class values, factors can be defined to operate on other factors, which enables the programmer to compose complex interactive computations from simpler ones. For example, we can define the function `map` as a function that yields a new (internal) factor, whose responses are generated by mapping a specified function over another factor as follows.

```
let rec map (f: 'b → 'c) (i: ('a, 'b) ftr) =
  ftr (fun (p: 'a) →
       let (h, i') = query i p in (f h, map f i'))
```

In the multiplayer game, this ability to create new factors in terms of others is used liberally. For example, simply but crucially, when communicating with the other user, we use the map function to append a newline character to the end of lines read from the `std_in` factor:

```
map (fun s → s ^ "\n") std_in
```

Note that the internal factor performs I/O indirectly by querying an I/O factor.

The above two examples considered synchronous queries, which block until the queried factor returns. Such behavior can be desirable—even necessary—in many cases. But sometimes it can be important to proceed without waiting for a response. For example, in Example 2, we wish to proceed executing the game loop even if the user does not press a key to be assigned to `my_key`, but we have access only to `key_presses`, an I/O factor that blocks until a key is pressed.

We therefore provide an *asynchronous query* operation, `aquery`, which takes the same arguments as `query`. This operation spawns a lightweight thread in which the factor is queried with the prompt. If the response is not available immediately, a *future factor* is returned. A future factor is a handle on the concurrently running computation. Like factors, future factors can be queried synchronously or asynchronously. However, since a future factor represents a running computation that has already been given a prompt, it cannot accept a new prompt. Thus, both `query` and `aquery` on future factors accept only unit prompts. Calling `query` on a future factor blocks until the original response is ready. Calling `aquery` on a future factor polls the computation. It will return the response and continuation if available, or the original future factor otherwise.

Example 4 (**Asynchronous query).** The current key of Example 2 can be obtained by asynchronously querying `key_presses`, which will return immediately even if no key is pressed (indicated by the constructor `Later`), in which case we fill in a space and continue.

```
let (my_key, key_presses') =
match aquery key_presses () with
| Now (k, key_presses') →
  (* key k is pressed *) (k, key_presses')
| Later key_presses' →
  (* no key is pressed *) (' ', key_presses')
in ...
```

Asynchronously joining factors. The primitives described so far turn out to be sufficiently powerful to express a rich class of interactive computations. In our empirical evaluation, we found a particular operation for asynchronously joining (merging) two factors into one, interleaving responses as they are available, to be very helpful. When queried, the joined factor asynchronously queries the two component factors and returns the first of the two responses. Even though such an operation is expressible based on the primitives described thus far, the resulting implementation is rather inefficient. We therefore include the primitive `ajoin` for joining two factors asynchronously.

Example 5 (**Asynchronous join).** In the multiplayer game, we create a factor for running the game loop, another, shown in Example 3 for writing chat messages from standard input over the network, and yet another to listen for incoming chat messages. Since these three operations should be performed concurrently, the implementation joins them asynchronously as follows:

```
ajoin gameloop
  (ajoin (output_on_network
           (map (fun s → s^"\n") std_in))
         (output_on_stdout
            (map (fun s → "[other player] "^s^"\n")
                 (input_lines in_channel))))
```

Splitting streams. Concurrency, as introduced by asynchronous joins, interacts poorly with the linearity requirement that factors be used only once. Suppose that, in our game, both the game loop and the outgoing chat thread depend on `std_in`.

```
ajoin (gameloop std_in) (map ... std_in)
```

We allow multiple uses of factors by permitting them to be *split*, which creates two handles to the underlying state or form of interaction. The above code can then be written as follows.

```
let (s1, s2) = split std_in in
  ajoin (gameloop s1) (map ... s2)
```

2.2 Algorithmic Semantics

We specify the semantics of the primitives algorithmically by using a pseudo-code notation based on Concurrent ML (CML) [39], which enables us to express the concurrent semantics.

In Concurrent ML, *threads* communicate by passing messages over *channels*. The call `spawn f` creates a new thread running the thunk `f`. A new channel is created using `channel ()`. Calling `send c v` will send value `v` over channel `c` and return unit. Calling `recv c` will receive a value on channel `c` and return it. Both functions are synchronous; sending on a channel will block until the value is received and receiving will block until a value is available. Non-blocking versions, `sendPoll` and `recvPoll`, exist as well, which return `None` if the corresponding blocking call would block. Finally, the calls `sendEvt c v` and `recvEvt c` do not perform message passing but return an abstract type of *event*. Events can be combined using `select`, which takes a list of events and blocks until the first of the events fires.

Figure 3 specifies the core primitives described in Section 2.1. The underlying data for non-future factors is a generator. Querying such a factor consists of simply applying the generator to the prompt. As implied by the informal description, a future factor need only be a handle to the concurrently running computation; in fact, it is a channel over which the response and continuation (type `('p,'r) view`) will be sent when finished. A synchronous query on a future factor is simply a (blocking) receive on this channel.

Asynchronous queries, like synchronous queries, apply the generator to the prompt. However, we do not wish to block while this call takes place. Thus, asynchronously querying a normal (non-future) factor spawns a new thread to complete the application. When `gen p` evaluates, the result will be sent along a new channel `c`. After the thread is spawned, it is immediately queried. (A small delay may be inserted here if desired to give the thread more time to be scheduled and complete.) If the function call returned immediately, the result is simply returned. Otherwise, a future factor is returned. A future factor can be asynchronously queried by again polling the channel. As described earlier, `split` is primarily a marker for the creation of aliases to factors. It just makes two new factors by duplicating the generator.

```
module I: Interactive = struct
  type ('p, 'r) view = 'r * ('p, 'r) ftr
  and ('p, 'r) ftr = 'p → ('p, 'r) view
  and ('p, 'r) fftr = (('p, 'r) view) channel
  type ('p, 'r) aview = Now of 'r * ('p, 'r) ftr
                      | Later of ('p, 'r) fftr

  let ftr gen = gen

  let query (gen: ('p, 'r) ftr) p = gen p
  let query (chan: ('p, 'r) fftr) () = recv chan

  let aquery (gen: ('p, 'r) ftr) p =
    let c = channel () in
    let _ = spawn (fun () → send c (gen p)) in
    aquery c
  let aquery (chan: ('p, 'r) fftr) () =
    match recvPoll c with
    | Some v → Now v
    | None → Later c

  let split (ft: ('p, 'r) ftr) = (ft, ft)

  type ('a, 'b) sum = Left of 'a | Right of 'b
  let ajoin f1 f2 =
    let (ic1, ic2) = (channel (), channel ()) in
    let (oc1, oc2) = (channel (), channel ()) in
    let rec f ic oc inj ft =
      let p = recv ic in
      let (r, c) = query ft p in
      send oc (inj r); f ic oc inj c
    in
    let _ = spawn (fun () → f ic1 oc1 Left f1) in
    let _ = spawn (fun () → f ic2 oc2 Right f2) in
    let rec gen is_comp1 is_comp2 (p1, p2) =
      let _ = if not is_comp1 then send ic1 p1 in
      let _ = if not is_comp2 then send ic2 p2 in
      match select [recvEvt oc1; recvEvt oc2] with
      | Left r → (Left r, ftr (gen false is_comp2))
      | Right r → (Right r, ftr (gen is_comp1 false))
    in
    ftr (gen false false)
end
```

Figure 3: CML specification of the core primitives of $\boldsymbol{\lambda}^{\mathbf{i}}$.

The function ajoin can be implemented in a straightforward way using repeated calls to aquery, but is a separate primitive in our library for improved efficiency. The primitive implementation, shown in Figure 3, spawns two threads, each with its own input and output channels. Each thread runs a loop that reads a prompt on its input channel, uses it to query the generator of one of the factors, sends the response over the output channel, and loops on the continuation. The body of ajoin then makes a new factor from a generator that sends its prompts to the two threads (if they are not already computing, as tracked by is_comp1 and is_comp2), and then waits for a response on either output channel. This response is returned along with a continuation that recursively calls the generator, updating the is_comp flags.

2.3 Formal Semantics

While it is not a focus of this paper, we have formalized the type system and the dynamic semantics of the sequential core of $\boldsymbol{\lambda}^{\mathbf{i}}$ and proved several metatheoretic results including type safety and the consistency of $\boldsymbol{\lambda}^{\mathbf{i}}$ programs with pure functional programming up to non-determinism. This work includes "on paper" formalization and proofs as well as a sizable full formalization in the Coq proof assistant. Details of this work are included in a separate technical report [34].

Statics. The type system for $\boldsymbol{\lambda}^{\mathbf{i}}$ enforces the single-use property of factors by using linear types [40, 7]. In the type system, I/O factors and variables that may be bound to I/O factors are held in separate contexts from standard (persistent) variables. In these *linear* contexts, contraction is not permitted and so each I/O factor or corresponding variable may be used at most once in a valid typing derivation[1]. Expressions that are closed under the linear contexts, and can therefore not refer to I/O factors, may be *promoted* to the persistent context and used without restriction.

Dynamics. The dynamic semantics of $\boldsymbol{\lambda}^{\mathbf{i}}$ follows the broad outlines of lambda calculus but models the interaction with the external world by using execution traces and user strategies [36]. Informally, a trace keeps a record of the program's interaction with the outside world by listing queries and splits. The user strategy models the stateful, non-deterministic behavior of the outside world by factoring out all effects using a deterministic function of the query and the current execution trace, which allows dependence on history.

Metatheory. We have used this formalism to prove several results. First, we established a type-safety theorem that proves that well-typed programs "don't go wrong." Second, we establish a correspondence between the imperative programs of $\boldsymbol{\lambda}^{\mathbf{i}}$ and purely functional programs. To this end, we convert $\boldsymbol{\lambda}^{\mathbf{i}}$ programs to purely functional programs by factoring out the non-determinism of interaction into a functional data structure provided as an input at the start of the program, and replacing queries to I/O factors with uses of this data structure. We show that the resulting program is equivalent to the original $\boldsymbol{\lambda}^{\mathbf{i}}$ program, and thus well-typed $\boldsymbol{\lambda}^{\mathbf{i}}$ programs may sensibly be viewed under a functional interpretation. We then use this result to show that, under some assumptions about the results of interaction, two $\boldsymbol{\lambda}^{\mathbf{i}}$ programs are equivalent if their functional conversions, described above, are equivalent. This shows that many of the same reasoning techniques one would apply to functional programs are applicable to well-typed $\boldsymbol{\lambda}^{\mathbf{i}}$ programs.

3 Implementation and Applications

3.1 OCaml Implementation

We have developed an implementation of the interface of Section 2 as an OCaml library. The implementation is built on top of OCaml's CML-style Event library and quite closely reflects the

[1]Because such variables are allowed to be used zero times, the type system may more precisely be called *affine*, but the use of the term *linear* for this sort of type system is common in the literature.

CML translation shown in Section 2. Since OCaml does not have a linear type system, however, our implementation cannot statically enforce the restriction that factors be used only once. We therefore check this property dynamically by furnishing the runtime representations of I/O factors with additional facilities that raise an exception when they are used more than once.

In addition to the small core language, our implementation includes a set of standard libraries with built-in I/O factors for interacting with the console and network as well as graphics and media libraries. The standard library also includes many useful functions, such as map, for building factors.

3.2 Applications

We describe some of the more interesting applications that we have implemented using the OCaml library, along with the features employed. All of these examples use concurrency in a non-trivial way, and many demonstrate the compatibility of factors with the full range of OCaml features, such as mutable references and foreign function interface (FFI).

Multiplayer Game. We implemented the motivating example of a multiplayer arcade game as described earlier. This code uses the graphics and network factors of our standard library to display and react to GUI events, and communicate between players over the network. The features of $\boldsymbol{\lambda}^{\mathbf{i}}$, for example using asynchronous queries to control polling and blocking, and using asynchronous join to create separate game and chat threads, are useful for this example. The game consists of 259 lines of OCaml code.

GUI Calculator. To test how well asynchronous join scales to many factors, we developed a simple GUI calculator, where each button consists of two factors, one which determines if the mouse is hovering over the button and one which determines if the mouse is clicking on the button, resulting in 38 factors to handle button events, all joined asynchronously. The arithmetic logic of the calculator is a factor which combines the values of the button-click factors and changes its state according to which buttons are pressed, producing numbers to be displayed on the screen. A separate factor draws the buttons, using information from the mouse-hover factors to shade a button if the mouse is over it. A screenshot is shown in Figure 4. The calculator involves 214 lines of code.

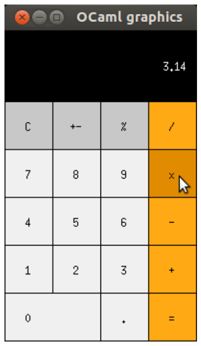


Figure 4: A simple GUI calculator

Physics Simulation. To test how well our techniques handle continuous calculations combined with user interaction, we developed a physics simulation involving two balls moving in a bounded 2D space on 2D trajectories. Each ball can be "free" or "caught." In the "free" mode, the balls move according to the laws of physics under gravitational force, bouncing off of the walls of the box and each other. For increased accuracy and efficiency, we vary the sampling frequency of the system time roughly linearly with the velocity of the balls (higher velocities lead to more sampling). By sampling infrequently when the balls are moving slowly, this policy ensures efficiency. By sampling frequently when the balls are moving quickly, the policy ensures accuracy. We further increase accuracy by predicting collision times (based on velocity and acceleration) and perform an update exactly at the time of a collision, ensuring that no collisions are missed. This example demonstrates the power of allowing sampling frequency to be determined dynamically at run time.

The user can "catch" a ball with the mouse and drag it. As a ball is dragged, its factor samples the mouse position in order to update the position of the ball, and also computes the velocity by continually taking the derivative of the mouse position, so that when the ball is released, the simulation can resume with the ball moving with the velocity at which it was released, allowing users to toss the balls around the screen. While a ball is caught, its integral computation suspends, and while both balls are free, the mouse position is not polled. Demanding input values only when they are needed reduces unnecessary polling and computation.

An asynchronously joined factor allows the user to change the color of the balls by pressing certain keys. This factor synchronously queries `key_presses`, which eliminates unnecessary polling and does not block other portions of the program because of the asynchronous join. This example totals 224 lines of code.

Streaming Music. We demonstrate the versatility of I/O factors by using them to encapsulate mutable state rather than I/O effects. In the process, we implement a streaming music server and client. The server opens a music file stored locally and streams it over the network using the network factors of our standard library. The client reads bytes from the network and plays the music, but to ensure that temporary network delays will not affect playback, a fixed amount of data is buffered on the client side. To make this work, two factors work asynchronously, sharing a mutable buffer whose implementation will be described below. One factor requests bytes from the network and writes them onto the end of the buffer. The other takes a number n as a prompt, removes n bytes from the start of the buffer and returns them. A third factor reads console input. These three factors are asynchronously joined into a factor that continuously fills the buffer, pauses and resumes playback based on commands typed into the console, and returns requested bytes from the buffer when the music is playing. This factor is then passed to a foreign function that allows the Simple and Fast Multimedia Library[2] to use the factor as an input stream to play the music.

We use I/O factors to manipulate the shared mutable buffer at a high level without the need for explicit synchronization. We use OCaml references to define a function `ref_ftr` which creates an I/O factor wrapping a new reference, initialized to a given value. Querying the factor performs an atomic read and update on the reference, as directed by a function passed as the prompt. Locking is handled by the code of the I/O factor. Splitting this factor corresponds to sharing the underlying mutable state, which can be done safely because operations are atomic. Operations performed concurrently may be interleaved arbitrarily, but the shared state will never be corrupted by partial reads or writes.

The (combined) OCaml code for the server and client consists of 163 lines of code. The function `ref_ftr` is an additional 11 lines of code, and the $\boldsymbol{\lambda}^{\mathbf{i}}$ sound libraries developed for this example consist of 160 lines of C++ code and 5 lines of OCaml code to perform the FFI.

[2]http://sfml-dev.org/

Unix Shell. As an example of a real-world program with many low-level interactions, we implemented `fsh`, a Unix shell that handles foreground and background jobs and supports history, command line editing and tab completion. If a foreground job is running, `fsh` periodically queries the factor `signals` to poll for signals from the operating system. Otherwise, `fsh` performs interaction with the user. To make sure that keystrokes will be captured while simultaneously monitoring for signals indicating that background processes have terminated, we asynchronously join standard input with the factor of signals and query this in a loop:

```
let rec inploop ... children input_and_signals =
  match query input_and_signals ((), ()) with
  | (Left c, is') → process_char c ... is'
  | (Right s, is') →
    let c' =
    if s = Sys.sigchld then reap children
    else children in
    (None, ftr (inploop ... c' aj'))
```

Low-level system operations are encapsulated within separate functions, while much of the code handles high-level operations, such as command line processing, operations on the data structures that store job status and command line history, and functions to support tab completion. These tasks are programmed quite naturally in the functional style of our library. The shell consists of approximately 500 lines of OCaml code, including extensive comments.

4 Evaluation

4.1 Expressiveness

The relatively broad range of applications that we implemented and described in Section 3 give empirical evidence about the practical expressiveness of the proposed approach. To establish the expressiveness with respect to prior work, we note that due to their inherent use of concurrency, the applications described in Section 3 would be difficult, if not impossible, to implement (without sequentializing) in existing implementations of FRP. But does that mean that our techniques are strictly more powerful? To answer this question in the affirmative, we implemented two existing FRP languages from prior work—Arrowized FRP and Elm—using our implementation of $\boldsymbol{\lambda}^{\mathbf{i}}$, in addition to futures, a powerful mechanism for concurrency (e.g., [20, 15]). We will briefly describe each embedding below and we present the full details of the embeddings in Appendix A.

Yampa. Yampa is a Haskell library based on Arrowized FRP [35], in which interactive programs are built up by combining *signal functions*, functions that transform signals. We have embedded a substantial subset of Yampa in $\boldsymbol{\lambda}^{\mathbf{i}}$, and used the embedding to implement the tailgating detection example of Nilsson et al. [35]. Our implementations of Yampa and of the tailgating example consist of approximately 300 and 120 lines of code respectively. We also directly implemented the same example in our $\boldsymbol{\lambda}^{\mathbf{i}}$ library in approximately 75 lines of code.

Elm. Our second implementation concerns the calculus called FElm that constitutes the core of Elm [11], which supports functional reactive programming. Unlike other FRP languages, FElm allows a form of asynchrony in which long-running signal-processing functions can be run asynchronously at the top level. The widely available Elm implementation, a compiler that targets Javascript, does not implement fully the asynchronous features of FElm because Javascript has limited concurrency support. We are able to implement all features of FElm directly using our library implementation of $\boldsymbol{\lambda}^{\mathbf{i}}$. The FElm embedding consists of approximately 70 lines of code (not including a custom queue implementation.)

Futures. As further evidence for the expressiveness of the concurrency features we propose, we show that futures can be encoded in $\boldsymbol{\lambda}^{\mathbf{i}}$, and thus the concurrency features of $\boldsymbol{\lambda}^{\mathbf{i}}$ are at least as powerful as futures. Creation of a future maps to creation of a factor (with unit prompt type) followed by an asynchronous query, and forcing a future maps to a blocking query. To represent futures faithfully, the representation also holds an already-computed value in the event that the asynchronous query returns a value immediately.

4.2 Performance

We have extensively tested each of the applications discussed in Section 3 and qualitatively evaluated their responsiveness. For example, the first author has used `fsh` as a primary shell for entire days without noticing a degradation in responsiveness, and two of the authors have demonstrated the multiplayer game by playing it together over the Internet from different locations.

Such tests and experiments, however, do not give quantitative information about the quality of interaction offered by an interactive application. Unfortunately, there is relatively little prior work on such quantitative performance evaluation. The primary reason appears to be that interactive programs can perform very small computations that are difficult to measure individually and reliably, especially in the context of a large number of interactions that must be tried out in order to obtain reliable measurements [14]. For example, Endo et al. [14] propose some techniques to perform such careful measurements but these techniques are fairly platform-specific, using features of Windows and Pentium hardware counters.

To quantitatively evaluate our proposed techniques in comparison with other possible approaches, we have developed an experimental framework for measuring the average responsiveness of an interactive program, as measured by its latency in responding to an interaction. Figure 5 illustrates the components of this framework, which enables generating a sequence of events, stored in a *trace file*, simulating an interaction between a program and its environment, and running an interactive program against this trace. Because a trace file, which consists of a (possibly very long) sequence of primitive actions (e.g. move the mouse, click, press a key, sleep), can be tedious to generate, we also developed a small domain-specific language for generating traces and implemented it as an OCaml library called `makeTrace`. Our top level script “runScript” takes a description of the trace, generates it, and runs a “driver” program with this trace, which replays the trace events in order to test interactive programs. The driver, a C program, takes as input a binary to evaluate and a trace file and forks a new process (with which it shares a pipe so the driver can mimic standard input), starts executing the target binary and performs the events from the trace file

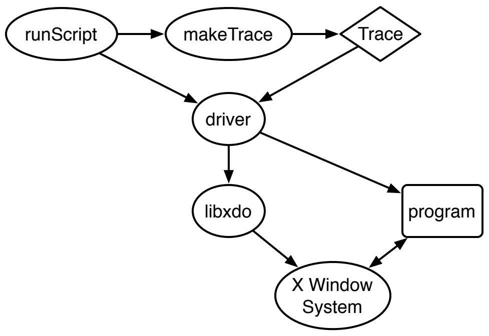


Figure 5: The components of the experimental setup.

in order, simulating mouse and keyboard actions through the X Window System using `libxdo`[3]. When finished with the trace, the driver kills the target program with a signal.

4.2.1 Benchmarks and Empirical Measurements

We developed a set of three (micro)benchmarks, each isolating a particular form of interaction. We consider and compare at least two implementations for each benchmark: one using our OCaml library and one written in OCaml with an event-driven style using effectful functions. For our first benchmark, we also implemented a version in Elm. Since Elm programs run inside the browser, precise quantitative comparisons between it and OCaml implementations are not particularly meaningful, but Elm is included in the first benchmark to exemplify our discussion of asynchrony. We choose Elm because it is widely considered to be the state of the art in usable FRP languages.

We performed all experiments on a commodity desktop with a 3.4GHz quad-core Intel® Core™ i7 processor and 8GB of memory, running Linux. OCaml's threads library works by time-sharing on one processor, rather than multi-processor execution [29], so only one core was utilized. For our experiments involving Elm, we used the Elm Javascript compiler version 0.12.3. To account for the fact that the compiled Javascript must run inside a browser, we ran the Elm trials in both Mozilla Firefox version 33.0 and Google Chrome 38.0.2125.122. We used the `pbench` benchmarking toolkit [4] to run experiments and collect data.

Counting clicks. This goal of this benchmark is to determine the responsiveness and correctness of an interactive computation that includes an interaction running concurrently with a slow computation. The benchmark creates two factors and processes them concurrently. Using the exponential-time recursive algorithm, the first factor computes the sequence of Fibonacci numbers, starting at `fib(0)`, with each element of the sequence initiated by a query. The second factor counts the number of (observed) mouse clicks. The benchmark asynchronously queries each factor and prints on screen the latest value of each factor. The aim is to count the mouse clicks correctly (without missing any) and responsively, while also performing the slow Fibonacci computation. The event-driven implementation creates separate threads for each computation. The

[3]http://www.semicomplete.com/projects/xdotool/
[4]https://github.com/deepsea-inria/pbench

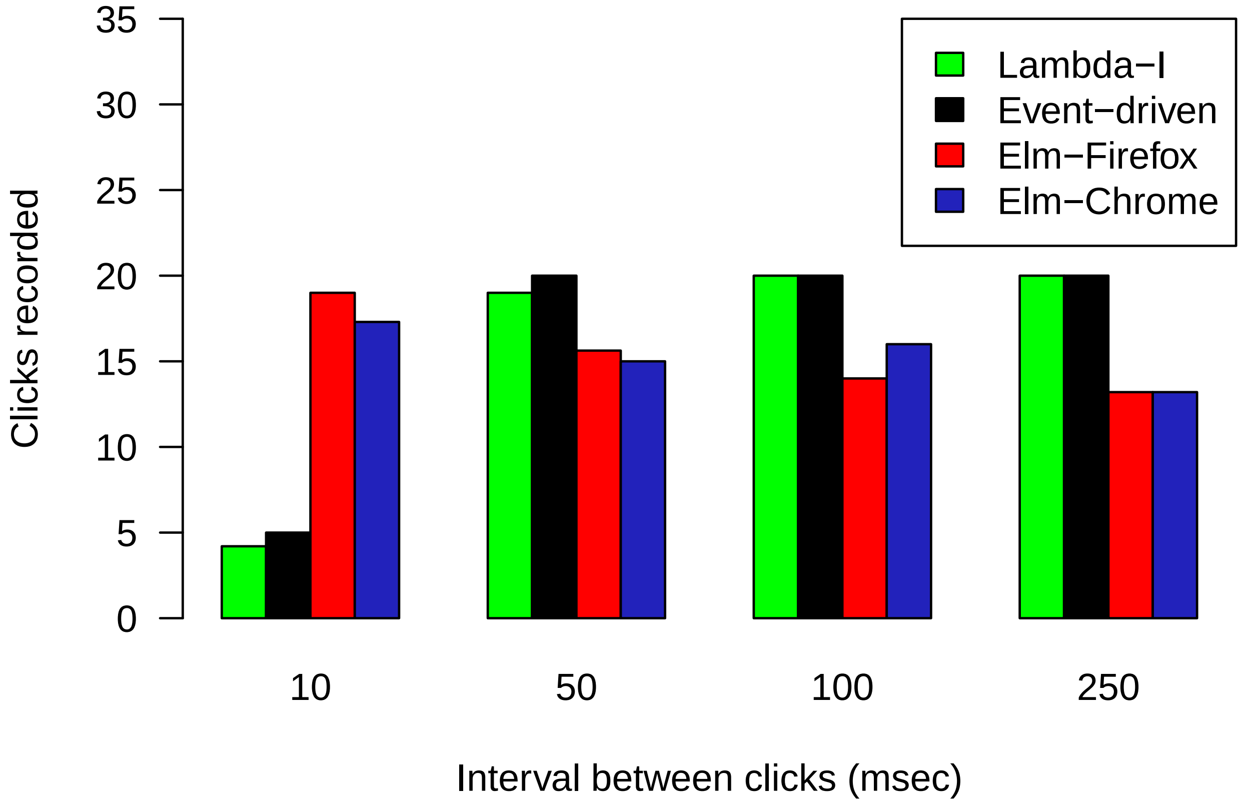


Figure 6: Number of clicks counted by each implementation versus click interval. The correct number is 20.

Elm implementation employs a single thread due to lack of concurrency support in the Javascript implementation of Elm. We will discuss the effect of this below.

In our experiment, we generate four traces, each of which simulates 20 mouse clicks, varying the interval between clicks from 10ms to 250ms. In each click, the mouse button is held for 100ms, resulting in a range of three to nine clicks per second. This range approximates the frequency at which a human might be able to repeatedly click the mouse. Figure 6 shows the average results over 5-10 runs of each trace. Each group of plots shows the number of clicks counted by the three implementations ($\boldsymbol{\lambda}^{\mathbf{i}}$, event-driven and Elm, in both of the browsers) on a particular trace. The x-axis label is the interval between clicks. Both the event-driven and $\boldsymbol{\lambda}^{\mathbf{i}}$ implementations perform poorly when mouse clicks are very fast. On all other traces, the event-driven implementation reliably responds to all clicks. By the time the frequency has dropped to 5 clicks per second, the $\boldsymbol{\lambda}^{\mathbf{i}}$ implementation also responds to all clicks. On traces where one or both implementations missed clicks, the two implementations performed similarly, differing by no more than one or two clicks on average.

The values for Elm show the effect that a lack of concurrency has on this benchmark. The Elm implementation is quite reliable on the faster traces, but appears to miss clicks on the slower traces. This is because, with slower traces, the long-running Fibonacci computations prevent Elm from processing the mouse clicks until a Fibonacci calculation ends. Thus, when the trace ends (and the experiment is stopped), any clicks done since the last Fibonacci calculation began are not recorded. If the Elm program is allowed to run longer, it eventually responds to all clicks. This example shows that, in the presence of possibly long-running computations, concurrency is vital to responsiveness.

Tracking mouse motion. The goal of this benchmark is to determine the precision at which continuous motion can be tracked while also performing some computation. The benchmark accomplishes both goals by computing the total distance traveled by the mouse using a standard numeric integration computation technique that approximates the integral by using rectangles. Since the accuracy of the integral improves by using smaller rectangles and since this happens only with quick reads of the mouse, we expect that the quality of the approximation quantifies the responsiveness of the computation.

To evaluate this benchmark, we generated four traces which moved the mouse in circles of radius 140 pixels with the period varying from 0.5 to 5 seconds. Each test was run for 5 seconds.

The results, averaged over 10 runs of each trace are shown in Figure 7. The rightmost bar of each group shows the correct distance for the number of circles the pointer will make at that speed in 5 seconds [5]. As expected, all implementations become less accurate at higher mouse speed. The performance of the $\boldsymbol{\lambda}^{\mathbf{i}}$ implementation is good, remaining within approximately 5% of the event-driven implementation.

Anagrams. The goal of this benchmark is to evaluate the case in which multiple sources of inputs (mouse and keyboard) are mixed with a relatively variable and potentially expensive computation. The benchmark displays the current mouse position as a pair of integers while finding English-language anagrams of words read from standard input (this was inspired by an example by Czaplicki and Chong [11], which involved translating words from English to French). For quality interaction, the mouse position should be updated frequently and standard input should be read and written promptly. To run both computations (mouse position and anagrams) concurrently, the $\boldsymbol{\lambda}^{\mathbf{i}}$ implementation uses two asynchronously-joined factors and the event-driven implementation uses two separate lightweight OCaml threads.

To quantify the responsiveness of the interaction, we measure the average number of times per second at which the mouse position is updated throughout a five-second run of the program. To prevent the mouse update thread from starving other threads and processes, including the anagram thread, the implementations place a delay of 0.0005 seconds between mouse updates. Thus the number of frames per second is limited to 2,000.

In our measurements, we performed 10 runs with each of four traces. Each trace supplied a single word on standard input, between six and nine letters, and then allowed the program to run for five seconds, with no additional words entered even if the anagram computation finished early. For the longest word, most of the program's run is spent computing the anagram, while the shorter words finished almost immediately. The results, plotted in Figure 8, show that the two

[5]This is calculated by summing the discrete distances moved by each command in the trace. We note that the correct distance cannot be computed by using a closed form formula (based on the circumference of the circle) because the travel distance is itself a rectangular approximation, where the rectangles are determined by discrete pixels on the screen

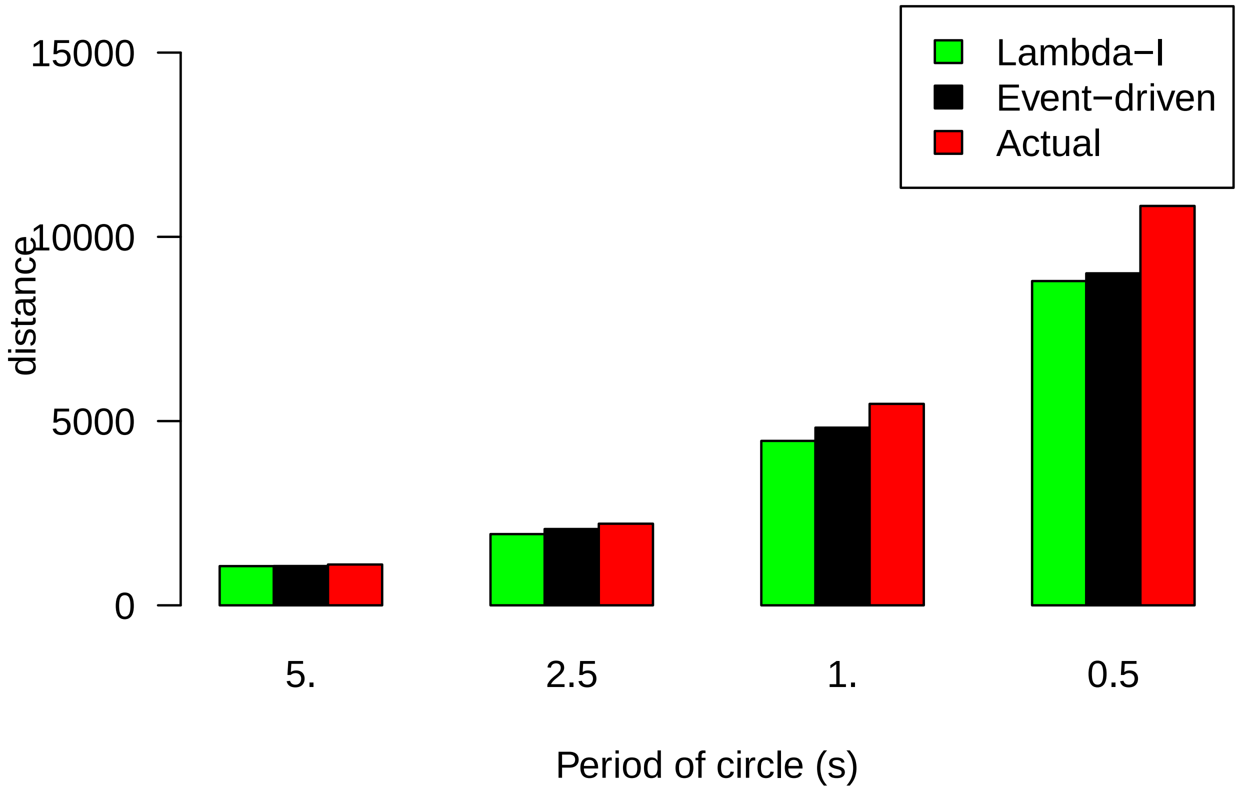


Figure 7: Total distance measured by each implementation versus mouse speed.

implementations are able to operate at similar frame rates, though both operate at only 60-70% of the maximum frame rate, even when taking the anagrams of short words. On the longest word, where the slow anagram computation uses CPU resources for most of the run of the program, both implementations achieve lower frame rates, though neither is affected disproportionately. The results also show that when the anagram computation requires time close to the allocated run time, the frame rate reduces by a factor of two, indicating that the two concurrent threads are able to time-share relatively fairly and efficiently.

5 Related Work

We discuss the most closely related work in the relevant sections of the paper. Here, we present a broader review of related work and its relation to this paper.

5.1 Event Driven Programming

Event-driven programming can lead to responsive interactive programs, but, as documented in the literature, this efficiency comes at the cost of a very low-level style of programming (e.g., [12, 16, 9]). Some prior research therefore proposed improvements to event-driven programming that can alleviate some of the complexities [16, 9]. Notably, the *responders* of Chin and Millstein [9]

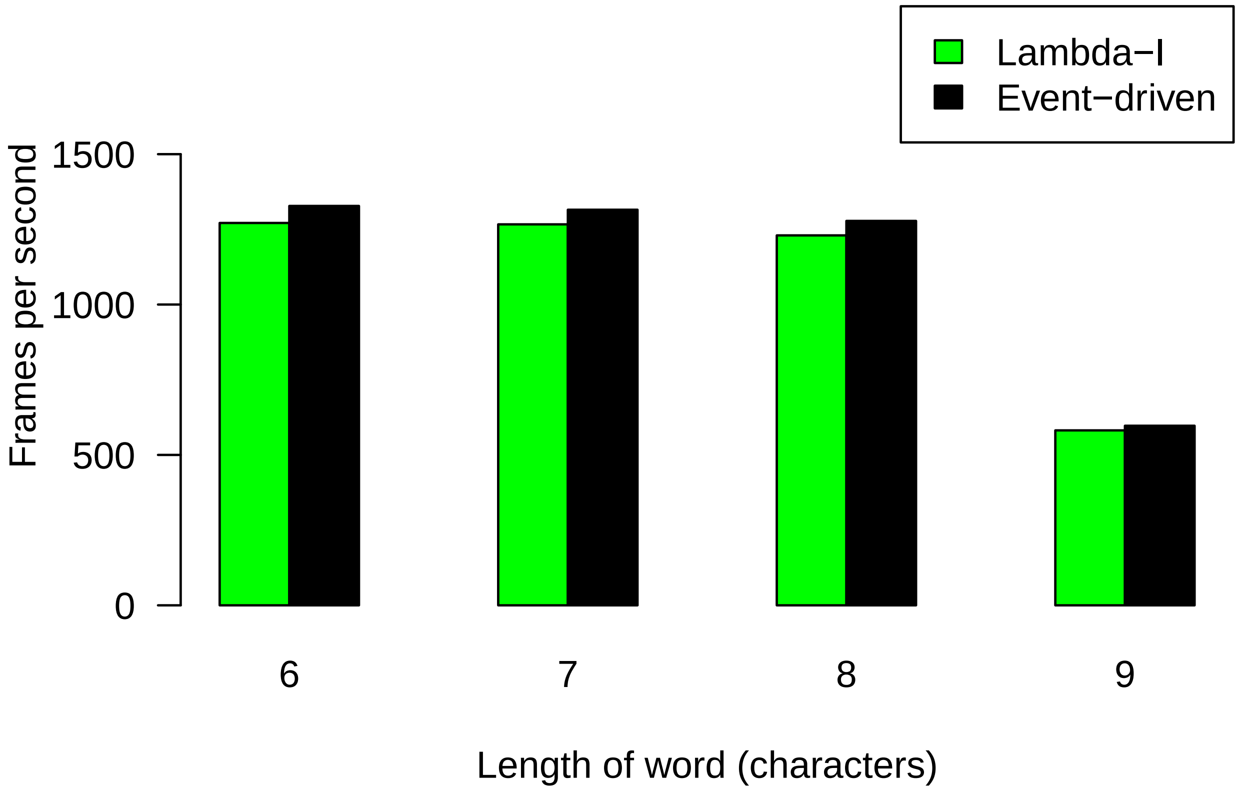


Figure 8: Average frames per second for mouse updates with different words.

share the idea of providing an abstraction that combines two-way communication with state. Chin and Millstein, however, do not consider concurrency, which is central to our work. In addition, our factors also differ from responders in that they are first-class values in a higher-order language (responders are class methods in an object oriented languages, Java). The query-response-continuation mechanisms of factors are in some ways similar to co-routines [25] but they are also different from co-routines, because the call structure is not symmetric.

5.2 Process Calculi and Concurrency

Process and concurrency calculi such as CSP [23], π-calculus [32, 33], and actors [22], have been proposed to model computations involving interaction between many processes via some form of communication medium, typically called "channels." Many variants of these calculi have been studied (e.g., several surveys exist [3, 21]), and several programming languages such as Concurrent ML [39] and Pict [37] have been designed based on these calculi. Our factors have some similarities to Plotkin's resumptions [38], which have an isomorphic type, but are used to model processes and non-determinism.

Unlike prior work on concurrency calculi, we consider the problem of programming a single process interacting with an external world, rather than modeling the interactions among many processes. This enables us to develop abstractions tailored to the interaction problem and avoid the full power and the complexity of concurrent programming. Specifically, we permit a process to use a specific, restricted form of concurrency related to implicit parallelism (described below) and can extend lambda calculus rather than concurrent calculi with our primitives. As we show, our approach can thus be used in the context of existing conventional multithreaded languages.

As it offers a restricted form of concurrency, our concurrency primitives are related to many others proposed in the past, including, for example, I-Vars [2], which allow separating computations into concurrent threads that can communicate synchronously.

5.3 Parallel Programming

As with popular parallelism abstractions such as fork-join [18, 27, 5, 24, 17, 28, 8] and futures [20, 15], our approach provides support for concurrency via a specific abstraction (asynchronous query), which is similar to futures in the sense that computations that do not complete immediately can be run in parallel. The key difference between futures and our concurrency constructs is that we allow concurrently running computations to be polled either synchronously or asynchronously, whereas futures can be polled only synchronously. In fact, as we show (Section 4.1), futures can be encoded in terms of our factors. We don't know if the converse is true but it appears difficult if not impossible because of the ability to poll asynchronously.

5.4 Functional Reactive Programming

We discussed the most closely related work on Functional Reactive Programming (FRP) in Section 1. More details and citations on FRP can be found in recent papers (e.g., [11, 26]), which span a broader scope than we can here. Our work is similar to FRP in the sense that our approach proposes abstractions for interaction that can be used as first-class values in a higher-order programming language. Specifically, use of functional programming languages is common to both approaches.

There are several differences between our work and FRP. First, our techniques are fundamentally concurrent, whereas nearly all work on FRP considers sequential programming. The only significant exception is Elm [11, 10]. While the theory of Elm includes a certain form of concurrency, its implementation is essentially non-concurrent. Second, while FRP abstractions are purely functional, ours are imperative (they can request input at run time), which we believe to be crucial to correctness, efficiency, and responsiveness of interactive programs. The third difference concerns expressiveness and generality: to avoid difficult efficiency problems called time and space leaks, nearly all FRP implementations—including Elm—restrict the set of allowable programs via language restrictions. Our approach does not suffer these problems and so does not need to impose these limits. Finally, the synchronous evaluation strategy adopted by FRP implementations leads to a range of previously recognized issues [11, 10] including the requirement to determine a clock rate for sampling that can guarantee correctness and responsiveness, which is technically impossible, because it requires computing tight upper bounds on run time. Our work allows the programmer to control the frequency of sampling, including sampling fully asynchronously.

6 Discussion

It is possible to give an imperative interface to factors by allowing them to maintain destructively-updated internal state instead of returning a continuation. This approach would somewhat simplify the language by eliminating the need for continuations, the `split` primitive and the linear type system. We avoid this approach for two reasons. First, the imperative interface complicates reasoning about interactive programs. Second, the continuation-based interface and linear type system make explicit the sharing (aliasing) of effectful computations, enabling the programmer to interpret the program purely functionally up to non-determinism (Section 2.3).

7 Conclusion

A important problem in language design and implementation has been the design of linguistic abstractions for expressing high-level and performant interactive programs. This paper presents a solution to this problem based on three ideas: 1) a single first-class data structure called a factor that abstracts interaction as exchange of information and internal change, 2) a synchronous and asynchronous query model for performing interaction, 3) a higher-order programming model that enables the programmer to write correct and responsive interactive programs at a high level. Our implementation and the examples considered show the techniques to be practically expressive and responsive. As part of our evaluation methodology, we developed techniques and software for evaluating interactive programs quantitatively, which may be of independent interest, and which we hope will lead to further empirical quantitative analysis of interactive programs and the development of other benchmarks and techniques.

References

[1] Elm language. http://elm-lang.org/.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.

[3] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005.

[4] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[5] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, 1987.

[7] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Carnegie Mellon University, April 2003.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[9] Brian Chin and Todd Millstein. Responders: Language support for interactive applications. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 255–278, 2006.

[10] Evan Czaplicki. Elm: Concurrent frp for functional GUIs, 2012. Senior Thesis, Harvard University.

[11] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 411–422, 2013.

[12] Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, 2009.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

[14] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 185–199, New York, NY, USA, 1996. ACM.

[15] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Brandeis University, Waltham, MA, USA, 1993. UMI Order No. GAX93-22348.

[16] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM.

[17] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[19] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag.

[20] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[21] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[23] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[24] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[25] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley, 1998.

[26] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without space-time leaks. *SIGPLAN Not.*, 48(9):221–232, September 2013.

[27] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[28] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, 2009.

[29] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.02*. INRIA, 2013.

[30] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, 2009.

[31] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, November 2007.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

[33] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, September 1992.

[34] Stefan K. Muller, William A. Duff, and Umut A. Acar. Practical and safe abstractions for interactive computation via linearity. Technical Report CMU-CS-15-130, Carnegie Mellon University School of Computer Science, August 2015.

[35] Henrik Nilsson, Antony Courtney, and Joh Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[36] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201, Piscataway, NJ, USA, July 2006. IEEE Press.

[37] Benjamin C. Pierce and David N. Turner. Proof, language, and interaction. chapter Pict: A Programming Language Based on the Pi-Calculus, pages 455–494. 2000.

[38] Gordon D Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[39] John H. Reppy. CML: a higher concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 293–305, 1991.

[40] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231 – 248, 1999.

[41] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[42] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10):146–156, 2001.

[43] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, 2002.

A Embeddings and Comparisons

A.1 Arrowized FRP

We show how to encode AFRP signal functions as factors in $\boldsymbol{\lambda}^{\mathbf{i}}$, allowing an AFRP library similar to Yampa to be implemented within $\boldsymbol{\lambda}^{\mathbf{i}}$. One implementation of AFRP [35] implements a signal function in approximately the following way.

```
type ('a, 'b) sf = time → 'a → 'b * ('a, 'b) sf
```

In this implementation, a signal function is a concrete function whose arguments are the amount of time that has passed since the last time the signal was sampled, and the current value of the input signal. The result is the new value of the output signal and the continuation of the signal function, which is ready to be used at the next time step. This type can be directly implemented using $\boldsymbol{\lambda}^{\mathbf{i}}$ factors.

```
type ('a, 'b) sf = ('b, time * 'a) ftr
```

Under this encoding, however, signal functions may only be used once, a restriction that doesn't exist in AFRP. To allow multiple uses, a signal function in our implementation must be *a function that produces* a signal function of the type above. This leads to the following implementation.

```
type ('a, 'b) raw_sf = ('b, time * 'a) ftr
type ('a, 'b) sf = unit → ('a, 'b) raw_sf
```

Standard AFRP signal functions and combinators can be programmed using this type. For example, `integral`, which in AFRP is simply a signal function that transforms real-valued signals[6] to their integrals, can be implemented as follows.

```
let integral () =
  let rec integral_gen a (dt, h) =
    let v = a +. dt *. h in
    (v, ftr (integral_gen v))
  in
  ftr (integral_gen 0.)
```

Note that `integral` is a function taking a unit argument, as required by our signal function type. To run top-level signal functions, we provide an outer loop that applies the given signal function to produce the underlying factor and then queries it continuously, calling a function `f` to handle each value in turn (`f` is likely an effectful function that, for example, prints the value to the screen.)

```
let run (f: 'a → unit) (sf: (unit, 'a) sf) =
  let rec run_rec s =
    let (h, t) = query s (delta, ()) in
      (f h; run_rec t)
  in
  run_rec (sf ())
```

Note that a refresh rate (`delta` above) must be specified and cannot be viewed or set by the program, though it could be varied dynamically by the runtime.

[6] In fact, the integral function allows integration of more complicated types, but we do not implement this here.

A.2 Elm

In Elm [11], programs are built from input signals and a small core of combinators. We represent Elm signals (and thus programs) as factors and have a toplevel loop that repeatedly queries the factor representing the program. To make this possible, the factors representing signals need a uniform prompt type. One idea would be the following:

```
type 'a signal = (unit, 'a) ftr
```

This leaves the question of how to handle input signals. If the same input signal is used in multiple parts of the embedded Elm program, the corresponding factor would need to be split and Elm has no notion of splitting. Furthermore, if two nodes of the signal graph request the value of an input signal on the same time step, for example if two buttons both request the mouse position, Elm's mostly synchronous semantics require that they get the same value, which the $\boldsymbol{\lambda}^{\mathbf{i}}$ semantics forbids. We thus pass all inputs as the prompt to all signals. The toplevel run loop accepts a factor which packages together all necessary I/O factors. This input factor is queried at each timestep, and the factor representing the Elm program is prompted with the input package, which the signal combinators then propagate to all signals in the signal graph.

```
type 'a signal = (input, 'a) ftr
val run : output signal → (unit, input) ftr → unit
```

Our implementation is a functor with the types `input` and `output` as parameters, along with a function `display : output -> unit` that prints or displays the output. In an Elm program, `output` would be a representation of the Elm type `Element`, which stands for an HTML document, and `input` would be a record containing all of the relevant input values (e.g. mouse position, keys pressed, time, etc.)

The toplevel loop is quite simple:

```
let rec run (p : output signal) (input : (unit, input) ftr) =
  let (inputs, input') = query input () in
  let (newval, c) = query p inputs in
  display newval;
  run c input'
```

With this representation in mind, the basic Elm combinators `lift` and `foldp` are easily expressed:

```
let lift (f : 'a → 'b) (s : 'a signal) : 'b signal =
  let rec lift_gen s inputs =
    let (h, s') = query s inputs in
    let h' = f h in
     (h', ftr (lift_gen s')
  in
  ftr (lift_gen s)
```

The function `lift` produces a factor that, when queried, passes the inputs to its argument factor, calls `f` on the response and responds with this value. The family of functions `liftn` for $n = 2, 3, ...$ are defined similarly.

```
let foldp (f : 'a → 'b → 'b) (e : 'b) (s : 'a signal) : 'b signal =
  let rec fold_gen s a inputs =
    let (h, s') = query s inputs in
    let a' = f h a in
     (a', ftr (fold_gen s' a'))
  in
  ftr (fold_gen s e)
```

The function `foldp` maintains an accumulator. When queried, it again queries its argument, passes the response and current accumulator to the functional argument, and responds with the new accumulator.

Elm's `async` combinator, which creates a new, asynchronous, branch of the signal graph, is somewhat more complicated and uses the concurrency primitives of $\boldsymbol{\lambda}^{\mathbf{i}}$.

```
let async d (s : 'a signal) : 'a signal =
  let rec gen s q cur input =
    let q' = Queue.push q input in
    match Queue.pop q' with
    | (Some input', q'') →
      (match aquery s input' with
      | Now (v, s') → (v, ftr (gen s' q'' v))
      | Later s' → (cur, ftr (genf s' q'' cur)))
    | (None, _) → failwith "Impossible"
  and genf s q cur input =
    let q' = Queue.push q input in
    match aquery s () with
    | Now (v, s') → (v, ftr (gen s' q' v))
    | Later s' → (cur, ftr (genf s' q' cur))
  in
  ftr (gen s (Queue.empty ()) d)
```

In Elm, all signals have a default value. In our embedding, the default value is only needed by `async`, which takes it as an explicit argument. The factor is defined using two mutually recursive generators. Both maintain the most recent value of the signal `s` as well as a queue of stored input values. The queue ensures that, even if `s` takes longer than a time step to compute, it will be passed the input values in sequence. One generator, `gen`, is used when the factor `s` representing the signal is a factor, and `genf` is used when it is a future factor. As such, `gen` retrieves the next set of input values from the queue and uses it as the prompt to `s`, and `genf` does not. Both asynchronously query `s`. If a value is available, it is returned and `gen` is called with an updated queue and current value. Otherwise, the maintained current value is returned and `genf` is called with the future factor.

A.3 Encoding of Futures

As described in Section 4.1, our representation of a future is either a future factor or an already-computed value.

```
type 'a future = Done of 'a | Fut of (unit, 'a) fftr
```

Creating a future from a thunk f involves defining a factor whose generator calls f (the continuation will not be used, and so is defined to simply be a recursive instance of the factor itself.) The factor is then asynchronously queried and the result, either a value or a future factor, is returned as the future.

```
let future (f: unit → 'a) : 'a future =
  let rec f' () = (f (), ftr f') in
  let i = ftr f' in
  match aquery i () with
  | Now (v, _) → Done v
  | Later i' → Fut i'
```

Forcing a future involves either returning the aleady computed value or synchronously querying (i.e. blocking on) the future factor.

```
let force (fut: 'a future) : 'a =
  match fut with
  | Done v → v
  | Fut i → let (v, _) = query i () in v
```