A WebAssembly backend for Links

CS M1 — Internship report

Loïc Chevalier

Année 2024-2025

Contents

1. Introduction

1.1. Links

Links [4, 3] is a research programming language, used to prototype different features of programming languages and see what kinds of constructions are useful for a programmer.

In particular, Links features *effect handlers*, a recent idea which extends the notion of exceptions. An exception is a language feature which allows a programmer to break the normal control flow of the program and raise an issue to an exception handler, allowing clean error handling. An effect extends this idea by also allowing the handler to resume execution to the handlee, giving some value to the effect. This can be used to implement threads or global states.

Links features two kinds of effect handlers, called *shallow handlers* (2.2) and *deep handlers* (2.2).

The `links` program, used to compile Links code, is written in OCaml. It features an interpreter, which allows running Links program as standalone programs or as a server, and a JavaScript backend, which allows compiling Links programs for web clients.

1.2. WebAssembly

WebAssembly [6] (or *Wasm* for short) is a new, W3C-managed, low-level programming language, first published in 2017 and targeted for the web. It is a stack-based, assembly-like language, with two file formats: a binary and a text format. There is an almost-one-to-one correspondance between the two formats, with the exception that all identifier names are usually absent from the binary format.

As a new programming language, it is still in active development and new features are regularly published. One such feature is the WasmFX extension [5], adding effects to WebAssembly. The effect handlers in this proposal (at the time of writing) is a third kind of effect handler: *sheep handlers*, from a mix of ***sh****allow* and *d****eep*** handlers. To resume from an effect, one must reinstall a handler (as for a deep handler) but the handler may change (as for a shallow handler).

1.3. My internship

The goal of my internship was to write a new backend to the `links` program, to translate from Links to WebAssembly, using this new WasmFX extension. I used the existing toolchain in the `links` program for that, adding a new backend.

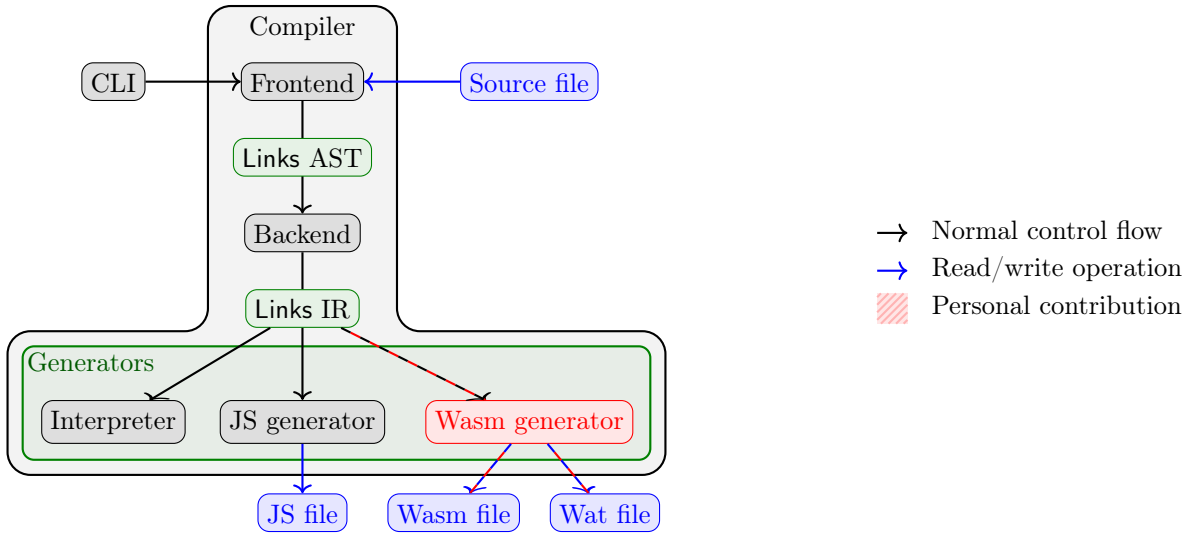
The entire code is available in the Links *GitHub page*, in the `wasm` branch.

2. The starting point

2.1. The Links toolchain

The Links toolchain is made of multiple files, which can be grouped into six rough categories:

- A command line interface;
- A frontend, which reads the input and parses programs into an AST;
- A backend, which transforms this first AST into successive intermediary representations, the last one of which I will call "the Links IR" in the rest of this document;
- An interpreter, which uses and interprets the Links IR;
- A Javascript (JS) generator, which generates Javascript code from the Links IR;
- A Wasm generator which I wrote, which generates Wasm or Wat code from the Links IR.



From the picture above, one can see that what I wrote does not really depend on the source Links language, but rather the Links IR. However, most Links IR constructs have their parallel in Links, so most examples will be given in their Links source code form.

2.2. The Links IR

The Links IR is inspired by A-normal form [2], and mostly follows the CoreLinks terms described in [3].

A *computation* represents a program or function, and is made of a list of *bindings* and a *tail computation*. A *binding* represents an assignment to a variable ID. A *tail computation* (see A.1) represents any "simple" computation, without any binding. Some very simple expressions (variables, constants, integer addition and so on) are also *values* (see A.1).

Bindings are further decomposed into:

- *Let-assignments*, with a (unique) binder and a tail computation;
- *Function declarations*, with a (unique) binder for the function name, one for each argument and en eventual one for a closure, and a computation for the body;
- *Alien declarations*, which are external functions (written in another language).

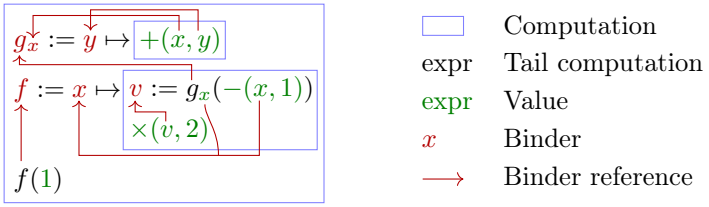
This IR has some type information, stored using the Church typing style [1], where only binders are typed.

During the AST to Links IR translation, there is also a step extracting inner functions to the global scope and generating the associated closure information, both at the definition and the call sites.

For example, the Links program

```
fun f(x) {
  fun g(y) {
    x + y
  }
  g(x - 1) * 2
}
f(1)
```

gets transformed into a binding defining `g`, one defining `f`, and a tail computation representing the impure call to `f` with the value `1` as argument. Furthermore, the binding defining `f` has a let-binding defining a temporary value `v` defined by the impure call to `g` with one value argument, defined as a pure function call (the integer substraction) between the values `x` and `1`; and the tail computation representing the return value is the pure function of integer multiplication called on `v` and 2. Notice that addition and subtraction, being operations that cannot fail or raise an effect, are considered values (pure function application); while the call to `g` is not (impure function application).



The Links language has multiple features. I will only describe the features I implemented from the Links IR in this section.

Core The core types are integers, string, lists, and row-polymorphic variants and records. There is also general polymorphism.

The core expressions are constants, variables and functions, with the following syntax:

```
sig f : (Int) ~> [| Ok : (a : Int) |] # Optional signature declaration
fun f(x) {
  # Returns an Ok with a record value
  # The record has a single field called 'a'
  Ok((a = x + 2))
}
f(1)
```

As said earlier, we can also nest function definition in the Links source file, which will become toplevel functions with closures in the Links IR.

Effects Another feature of the Links language is effects. As said in the introduction, an effect system is an extension of an exception system: effects interrupt the normal control flow of a program like exceptions, but they can be *resumed* or *continued* from the exception point. Both verbs will be used interchangeably in the rest of this document.

Effect signatures are a part of the typing system. Every effect is identified by a name; one effect name may have multiple signatures depending on the location. For example:

```
fun log(msg) {
  do Log(msg) # Raise the effect 'Log' with 'msg' as the argument
}

sig log_string : () {Log : (String) => a|_}~> a
fun log_string() {
  log("Hello") # Raises Log("Hello") (by calling 'log')
}

sig log_int : () {Log : (Int) => a|_}~> a
```

```
fun log_int() {
  log(42) # Raises Log(42) (by calling 'log')
}
```

`log_string` raises the effect `Log` with a string, while `log_int` raises the same effect name with an integer. The type system then makes sure that different effect signatures with the same name cannot appear at the same location.

To receive an effect, we use the `handle` (or `shallowhandle`) construct:

```
handle (f()) {
  case n -> Output(([], n))
  case <Log(msg) => k> ->
    # Resume, collect the rest of the logs, then prepend msg
    switch (k()) {
      case Output((rest_of_log, ret_val)) ->
        Output((msg :: rest_of_log, ret_val))
    }
}
```

They are defined by the handlee computation, a normal return case binder and computation, and a list of effect handlers (4-uples of the effect name, one binder for the content, one for the continuation and a computation). The difference between `handle` (*deep handlers*) and `shallowhandle` (*shallow handlers*) appears when the continuation raises an effect. If you raise an effect in a continuation, if the handler is deep then the same handler will catch said effect; if the handler is shallow the effect will be passed to an earlier handler. Note that in this context, returning a value to the handler is also "raising an effect".

In other words, these two programs are equivalent:

```
handle (f()) {
  case n -> ([], n)
  case <Log(msg) => k> ->
    var (msgs, n) = k();
    (msg :: msgs, ret_val)
}
```

```
fun handle_log(f) {
  handle (f()) {
    case n -> ([], n)
    case <Log(msg) => k> ->
      var (msgs, n) = handle_log(k);
      (msg :: msgs, ret_val)
  }
}
handle_log(f)
```

Actors The actor system is an other control flow system, which predates effects in Links. It is a form of lightweight thread system.

Each actor is a thread. Each actor also have a *mailbox*, to which actors can send messages. The mailbox owner can then receive those messages in a FIFO ordering. The Links interpreter then needs to preemptively schedule these actors, simulating a multithreaded application.

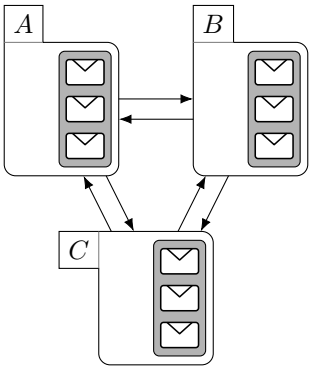


Figure 2.1.: Representation of three actors

In Links, we have the following syntax:

```
var root = spawn {              # Spawn a new process
  var some_computation = f();
```

```
  receive {                    # Check the mailbox
    case SendAnswerTo(proc2) ->
      proc2 ! ReceiveAnswer(some_computation); # Send a message
  }
};

root ! SendAnswerTo(self()); # Get the actor's own representation
                             # then send it to the "child" actor
var some_other_computation = g();
```

This program spawns a new actor, which then runs the function `f()`. In parallel, the parent actor sends a message, so once the computation in the child actor is done, it is transmitted back to the parent using another message. Note that `f()` and `g()` are computed in parallel.

The high-level constructs are transformed into calls to the low-level functions `recv`, `Send` and `spawnAt` in the Links IR; no trace of these high-level constructs remain.

The construction `spawnWait` (and its low-level function `spawnWaitAt`) is similar, but the parent waits for the child to return a value; the expression then evaluates to that value, instead of the new actor representation.

Finally, there are special actors, called *angel actors*, that are similar to regular actors except that the program will not terminate until all angel actors exit. The initial actor can therefore be seen as an angel actor. They are created using the `spawnAngel` construction (and its low-level function `spawnAngelAt`).

2.3. WebAssembly

As described in the introduction, WebAssembly is a stack-based, assembly-like programming language. As such, it only has a few low-level features. It is also a type-safe language, with various type annotations to enable easier type-checking (which is a part of the process called *validation*).

WebAssembly code is a list of instructions packaged in functions. A WebAssembly instruction may push to and pop from from the value stack, and can be to load or store a value from the local values (which include the arguments), replace the two topmost (integer) values in the stack by their sum, and so on.

There is no general `goto` instruction; instead, there are structured instructions such as `loop` or `block`, which you can branch to from inside (for example, using the `br` instruction) and which will move the instruction pointer to either the beginning of the instruction (in case of a `loop`) or just past the end (in case of a `block` and `if`). These structured instructions are parameterized by a function type, describing their effects on the stack, which allows for efficient type-checking.

Arguments to functions and intrinsics are also passed through the stack, and the return values are placed on the stack. Values and blocks are typed, where a type is:

- A base numeral type, either integer or floating-point number, and either 32 bits or 64 bits;
- A function type, with a list of arguments and return values;
- A reference to another type or to anything (a "top type").

There are two kinds of files: text files, with the `.wat` extension and written using S-expressions, and binary files, with the `.wasm` extension. Both formats are easily compilable into the other in an almost one-to-one manner, the only exception being that local names are not stored in the binary format.

An extension, called the GC extension (for *G*arbage *C*ollector), has been proposed to, among other things, add structures and a top type for structures, allowing the use of references to any structure.

The WasmFX extension, also known as *typed continuations* or *stack switching*, includes the GC extension and adds *tags*, which can be understood as effect labels and provide the effect signature; *continuations*, which can be seen as the set of information needed to resume a computation; and some instructions, among which `resume`, which installs an effect handler and starts or resumes a continuation, and `suspend`, which halts a continuation with an effect (specified by the instruction) and yields some value(s) to the corresponding effect handler, according to the signature associated with the tag.

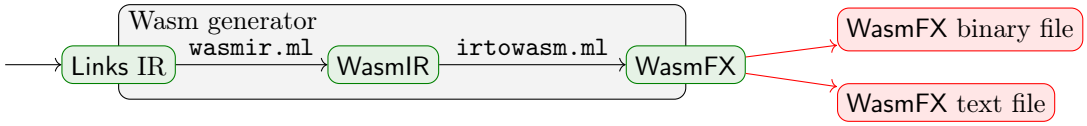
Tags are global compile-time constructs, declared at the module level (thus they necessarily have a global scope). The `suspend` instruction then has a tag parameter, which specifies which tag must be used for the suspension, and the `resume` instruction has a (potentially empty) list of tag handlers which specifies which tag suspensions must be handled by it and how.

New continuations are created from function references, and corresponding continuations are generated when catching an effect.

3. The WasmFX generator

My work can be split in two parts:

- Transform the existing Links IR into what I called the *WasmIR* (see A.2);
- Transform this WasmIR into Wasm code with the WasmFX extension.



A Links computation gets mapped to a WasmIR module, which then gets mapped to a WasmFX module.

A WasmIR module can be seen as a non-empty list of function-like object definitions (functions, handlers and builtins). Each function is a list of assignment and a final expression. Each handler is a list of effect handlers and a finalizer.

The WasmIR was written using GADTs, which enables the compiler to check for obvious typing errors.

Both translations use two enviroments: a global environment, implemented in the respective `GEnv`, and a local environment, implemented in the respective `LEnv`. The global environment is always available, and stores information about the number of functions, information about the effects used, and so on; while the local environments (at least one for each function) are only available in functions and store information about local variables, the number of arguments, and so on.

The global environment of the WasmIR to WasmFX translation also has a type environment, implemented in the `TEnv` module. This module contains every information about types, and the more generic global environment only has a reference to that type environment.

In WebAssembly, functions are numbered by their index in the list of defined functions, including imports. This means that imported functions always have IDs smaller than defined functions.

3.1. Core features

The compilation of the basic core features is straightforward: integers and strings are mapped to integer and strings in the WasmIR; lists of α and variants are mapped to generic lists and variants in the WasmIR; type variables are mapped to the abstract WasmIR type. Records are also mapped to n-uples. Note that the current version does not support polymorphic records (polymorphism at the field level, not just the type of the fields). Runtime integer and boolean values are *unboxed*: they are implemented by `i64`s and `i32`s (more on that later). This means they can be created and used with zero cost, but extra care is required to support polymorphism.

Both Links IR tail computations and values are mapped to WasmIR expressions. Computations are then mapped to a list of assignments and a final expression.

The core tail computations, apart from function-related computations, have an intuitive mapping to expressions. Note that no expression can manipulate the content of values with the abstract type.

In order to unify the calling convention, all functions must be closed by the `EClose`, `ERawClose` or any `E*Handle` expressions, which all define a closure to pair with the function closure, before being used elsewhere. The type of those expressions is an instance of `TClosed`. On the other hand, there is a single `ECallClosed` expression taking the closed function and a list of arguments, which calls the underlying function with the arguments and the associated function closure.

Polymorphism is also managed at the function level, as this is the only place where polymorphic instantiation can occur.

Indeed, some Links functions may expect polymorphic variables while they are called with concrete values. This is similar to how one can define in OCaml the identity function `val id : 'a. 'a -> 'a` which can then be used by replacing `'a` with any type. In this case, we need to *box* and *unbox* the runtime value given to and returned by the function. This is managed in the WasmIR through the addition of `ESpecialize`, which take in a closed function and a type variable specialization (i.e. a partial mapping from type variables to other types) and gives a closed function with

the specialization applied. This expression would transform the `id` OCaml function above with a function with a fixed signature `int -> int` or `'b -> 'b`.

In the WasmIR, this is translated using the fact that a closed function type is described by the types of the arguments and the type of the return value, but also by how many type variables are abstracted at the front, with the type `('g * 'a -> 'b) typ` (where `'g` represents the number of abstractions, `'a` the arguments type list and `'b` the return type). For example, in the OCaml `id` function above there is one toplevel type abstraction `'a`.

The `ESpecialize` constructor then transforms a `('gs * 'as -> 'bs) typ` into a `('gd * 'ad -> 'bd) typ` using a `('gd, 'gs) specialization` (specifying the number of abstractions we removed) alongside a `('ad, 'as) box_list` (describing how to box the arguments) and a `('bs, 'bd) box` (describing how to unbox the result). The function call expression `ECallClosed` then requires that the number of type abstractions `'g` of its argument is zero (described by `'g = unit`). This is similar in OCaml to how we need to specify what `'a` and `'b` are in the context of the caller before we can call the underlying function, even when said underlying types also contain type variables (present in the context of the caller).

The translation from WasmIR to WasmFX is mostly straightforward for the core Links features. Notably, integers are implemented using 64-bits integers.

The only harder choice to make was choosing the implementation of the abstract WasmIR type. I chose the `ref null eq` top type, which is the supertype of all structures and arrays; value boxing and unboxing therefore have a low cost.

All Links variant types are mapped to a uniform variant type of a pair of a tag and a boxed value. Similarly, values in a list are always boxed, and a list is represented by a nullable reference to a pair of the head and the tail (which is a list). The empty list is a null reference.

A direct call to a function in the WasmIR will be compiled into a direct call in `WebAssembly`. However, functions are not directly used as callable objects, due to the need of a homogeneous representation for n-ary functions because of polymorphism. We instead use a triplet ($\texttt{call}_f, f, \texttt{closure}$) of two function references and a closure to represent a callable value. To call one, we call the first element of the triplet (the function reference) with the boxed arguments first, then the second and third element of the triplet (the other function reference and the closure content) last. If the function takes no closure, the callable value will be constructed with an empty (null) closure and the function will ignore this closure argument; the argument will however remain present, to unify their representation.

```
f(5)
```

```
(i64.const 5)
(ref.null none) ;; if f does not have a closure
(call $f)
```

```
vf(x) # vf variable
```

```
(local.get $x) ;; assume x already boxed
(local.get $vf)
(struct.get $vf_type 1) ;; f
(local.get $vf)
(struct.get $vf_type 2) ;; closure
(local.get $vf)
(struct.get $vf_type 0) ;; call_f
(call_ref $callf_type)
```

3.2. Effects

Handlers are more complex. Due to the fact that handlers in WebAssembly are neither deep nor shallow handlers, we need to extract handlees into new functions. Furthermore, every deep handlers also needs to be extracted into new functions, as we need to reinstall the same handler when resuming.

Therefore, the following Links handler expression:

```
handle (f()) {
  case v -> [v]
  case <Log(i) => k> -> i :: k(())
}
```

gives two extra functions (one for calling `f()`, one for the rest of the handler) and the expression it is mapped to is an `EDeepHandler` expression with reference to both of these functions.

To do so, I used three kinds of local environments in the first translation: there is the main local environment (without arguments nor closure), used for the toplevel computation translation; there is the normal local environment (with arguments and a closure), used as the local environment of functions; and there is the local sub-environment.

This sub-environment holds a reference to a parent local environment and a working closure. Whenever a variable lookup for some variable v occurs in the sub-environment, the following process is executed:

1. If v is found in the sub-environment, this variable is returned.
2. Otherwise, v is searched in the parent environment.
3. If the search returned a local variable in that parent environment v_p, a new local variable v_ℓ is added to the closure for the sub-environment. A new closure initializer mapping v_ℓ to v_p is added. v is then mapped to v_ℓ in the sub-environment, and v_ℓ is returned.
4. Otherwise, the result is passed as-is.

For example, in the following code:

```
var x = 1; var y = x; var z = x;
handle (x + y - 1) { case x -> var y = z; x - y }
```

the handlee `x + y - 1` gets extracted into a standard function requesting a closure containing `x` and `y`, and the rest of the handler `handle . { case x -> var y = z; x - y }` gets extracted into a handler function requesting a closure containing `z`.

Indeed, the first computation requires a lookup of both variables `x` and `y`, but the second computation requires a lookup of the variable `z` found in the parent environment, and of variables `x` and `y` both found in the sub-environment directly.

A handler expression such as

```
handle (f()) {
  case v -> [v + 1]
  case <Log(i) => k> -> i :: k(())
}
```

then gets translated with the following steps:

1. The code contains a function c_f which calls `f`. It expects a boxed argument (which will be ignored), and will return an `i64`.
2. It also contains a function h, which will be the handler itself. It expects one (empty) closure v_1, a continuation v_2 expecting a boxed argument and returning an `i64`, and a boxed argument v_3. h will return a list (of `i64`, though this is not encoded).
3. First, create a new continuation κ_0 with c_f as the base function.
4. Call h with an empty closure, said continuation κ_0 and a null reference.
5. In h, resume the continuation argument v_2 with the boxed argument v_3 with a handler for `Log`.
6. If v_2 returned normally, add 1 to the result, and create a one-element list with the result.
7. If v_2 generated the effect `Log`, it also provided an argument v_c and a new continuation $\kappa_{\texttt{Log}}$. Recursively call h with v_1, that new continuation $\kappa_{\texttt{Log}}$, and an empty record. Then, prepend v_c to the return value of the recursive call, and return this new list.

The fact that the Links handler is deep is encoded by the recursive call to h in step 7. With a shallow handler, one would instead simply resume the new continuation directly, without installing new handlers.

3.3. Actors

Actors induce a non-trivial cost. To avoid paying this cost even when actors are unused, the use of the low-level functions related to actors is detected during the first translation step. The actor implementation is therefore only added if it is necessary.

The implementation of actors is split in six WasmFX operations:

1. Spawn an actor, and get its actor representation;
2. Wait for an actor to finish (using its representation);
3. Register an actor representation as an angel actor;
4. Get the active actor representation;
5. Terminate the program;
6. Yield.

The last operation is added because WasmFX does not support concurrency; cooperative scheduling is therefore required.

An actor representation is a pair of an actor UID (an integer) and an actor representation content, as shown in figure 3.1. This content may be a boxed value, if the actor finished executing; or it may be a quadruplet (b, m_i, m_o, w), where:

- b is a boolean (`i32`) showing whether the actor is blocked.
- m_i and m_o are two messages lists, implementing an optimized FIFO queue: messages are pushed to m_i and popped from m_o when m_o is non-empty. If m_o is empty, m_o gets replaced by the reverse of m_i, and m_i gets set to the empty list. If m_o is still empty, b is set to true and the actor yields. The blocked state gets unset when an actor pushes a message.
- w is the *waiting list*: a list of actors that request the return value of the actor, once it finishes.

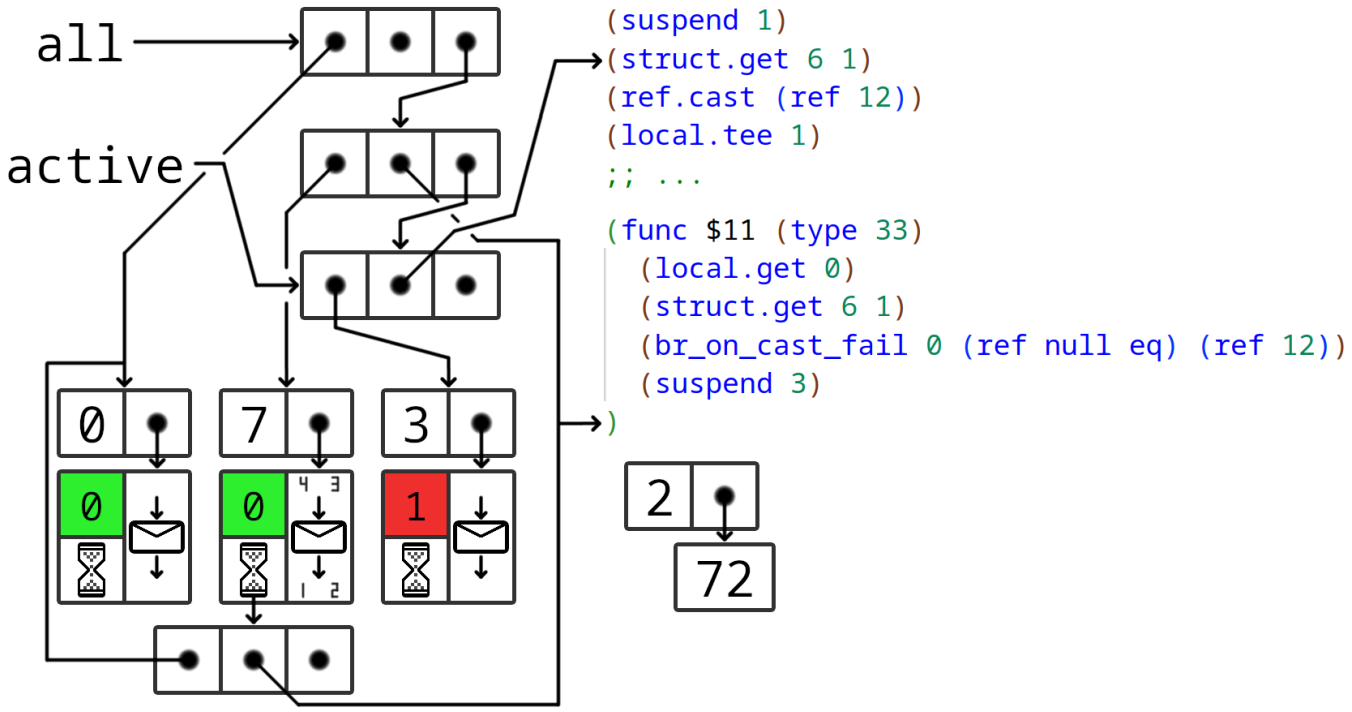


Figure 3.1.: Example of a program state at some point during execution

An actor can therefore be in multiple states:

1. it may have finished (the representation has a boxed value);
2. it may be waiting on another actor (the representation is in a waiting list);
3. it may be blocked (the b flag is set and the representation is in the main list);
4. it may be ready (the b flag is unset and the representation is in the main list);

5. it may be active (same as the ready state, and it is pointed by the active pointer).

All blocking, ready and active actors have their representation in a linked list along with a corresponding continuation. A pointer to the active actor (if there is one) in that linked list also exists.

Notice that the state of an actor is entirely implicit. Indeed, in figure 3.1:

- Actor 0 is in the waiting list of actor 7, so is in the waiting state;
- Actor 2 has finished with the value 72;
- Actor 3 is blocked, and may be the active actor until the next `yield` effect;
- Actor 7 is ready, and has four messages in its mailbox.

Note that due to implementation details, actor representations in the main linked list with no continuation (and are therefore in the finished or waiting states) should be removed from the list, but are only removed during a later scheduling. This is the case for actor 0 in the figure.

When actors are detected, the main function changes; the function which computes the program output gets extended to wait for all angels to finish, then emits the `exit` effect. The main effect also changes from that function to a new function, which provides scheduling capabilities.

This scheduler function installs handlers for every actor effect needed: `exit`, `self`, `spawn`, `wait` and `yield`.

The `yield` effect pauses an actor and the scheduling gets executed to find the next actor to run. This includes an overhead; to avoid increasing the overhead too much while still never starving any actor, a yield operation is executed before each `call`. This operation decreases a global counter, and only once this counter reaches zero is the `yield` effect raised. The counter is also reset after every scheduling operation.

One more feature of the WasmFX extension is the control flow instruction `switch`.

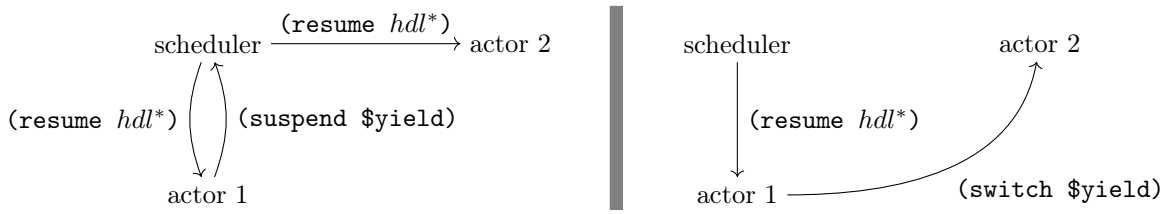


Figure 3.2.: `suspend` versus `switch`

Instead of switching the active process to the scheduler, performing the scheduling operation, then `resume`-ing the next process with the same handlers, we can simply perform the scheduling operation in the active process then `switch` directly to the next process, without switching to the scheduler.

This may be useful, since a context switch has an inherent cost to it. The version using `suspend` therefore has two context switches per `yield` effect, whereas the version using `switch` only has one. However, there is more boilerplate required to implement `switch` compared to `suspend`, so the actual speed benefit will depend on the engine used among other things.

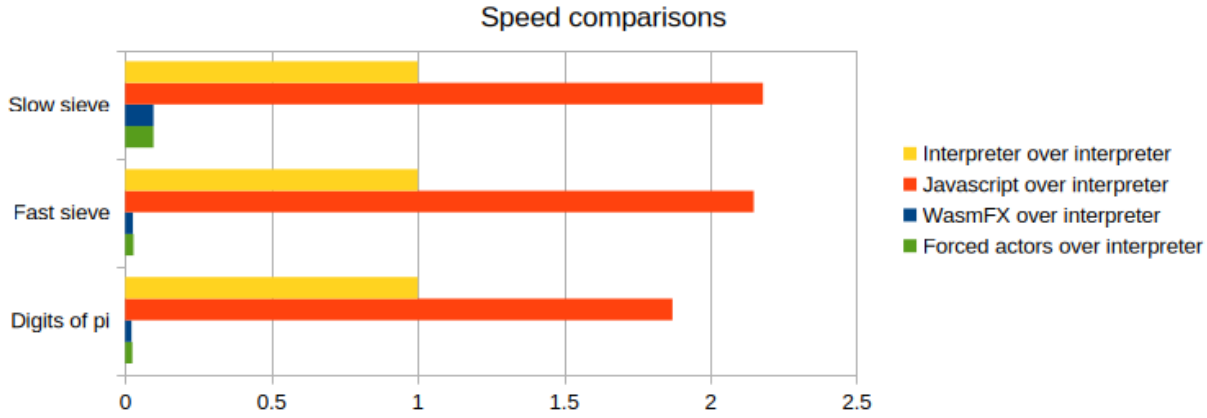
4. Results

I used three test programs to evaluate my work (all available in the appendix B): an algorithm computing the first 800 digits of pi, an implementation of Eratosthenes' sieve algorithm to find all prime numbers less than 20000, and a variation on this algorithm that uses modulos (the "variation on sieve" program). Unfortunately, Links does not support arrays, so the true implementation is slower than the variation.

I used the `links` program without argument to interpret the IR, with the `-mode compile` option to generate a Javascript program that can run in the web browser (here, Chromium with the V8 engine), and with the `-mode wasm` option to generate a WasmFX program that can run with Wizard. I then ran all tests 50 times with the Wizard engine, 10 times with the interpreter and 5 times with the web browser (since those tests are much slower individually). I also, in a separate run, manually forced the support for actors in the WasmFX implementation. This does not impact the variation on the sieve program since it already uses actors.

The results in the table are given in seconds. The results in the graphic are divided by the time the interpreter took.

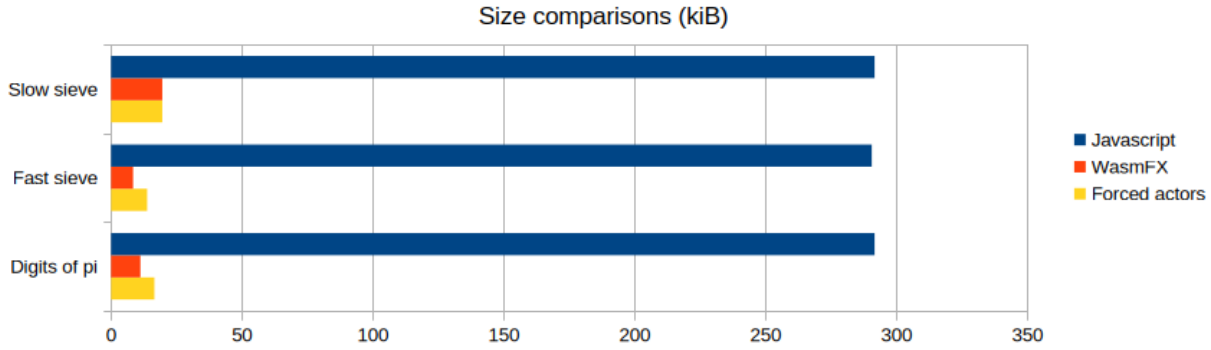
Program	Interpreted	Compiled to Javascript	Compiled to WasmFX	Forced actor support
Variation on sieve	24.14	52.630	2.3369	—
True sieve	148.94	319.985	3.951	4.2496
Digits of pi	4.093	7.657	0.0884	0.1005



My work can therefore execute Links programs about ten times faster than the interpreter, and about twenty times faster than the Javascript backend. Forcing the actor support therefore only decreases the speed by about 10%.

The following table summarizes the size of the byproducts. Note that the interpreter has no byproduct. The graphic below only shows the size of the binary format.

Program	Javascript file	WasmFX	Forced actor support
Variation on sieve	6849 LoC 285 kiB	813 LoC / 20 kiB (text) 2.2 kiB (binary)	—
True sieve	6819 LoC 284 kiB	357 LoC / 8.3 kiB (text) 969 B (binary)	561 LoC / 14 kiB (text) 1.5 kiB (binary)
Digits of pi	6849 LoC 285 kiB	478 LoC / 11 kiB (text) 1.3 kiB (binary)	678 LoC / 17 kiB (text) 1.8 kiB (binary)



The massive size difference between the Javascript files and the Wasm files is due to the Links prelude. The Javascript backend includes the entirety of the prelude (which contains more than 300 functions), while the Wasm backend does dead code elimination from the prelude (though not from the main program), therefore only including what is really needed.

From the results above, forcing actor support also increases the binary file size by about 50%.

5. Conclusion

I fulfilled my internship objective, which was to write a WebAssembly backend for Links. This improves the WasmFX ecosystem, which is a very recent extension to the WebAssembly language and thus is very lacking (both in terms of producers, ie. compilers, and in terms of consumers, ie. interpreters).

The next steps of the Links compiler would be to add support for more builtin functions, such that a replacement of either the Javascript backend (used by web clients) or the interpreter (which can be used as a web server) by the WasmFX backend would be possible. What is lacking for this is mostly HTML- and databases-related functions, which are tangential to the effect system (thus not what my internship was focused on), and a support for multishot handlers (the ability to resume from the same suspension multiple times in one handler).

My work has also been used by an other M1 student who took an internship at the same time as me to support multishot handlers in the WasmFX backend.

In the end, my work is a starting point in exporting Links programs to WebAssembly, the new architecture-independent programming language of the Internet.

A. Appendix: syntaxes

A.1. Links IR syntax

```
type tail_computation =
  | Return      of value
  | Apply       of value * value list
  | Special     of special
  | Case        of value * (binder * computation) name_map
                   * (binder * computation) option
  | If          of value * computation * computation
and special =
  | Wrong       of Types.t
  (* ... database manipulation constructions ... *)
  | CallCC      of value
  | Select      of Name.t * value
  | Choice      of value * (binder * computation) name_map
  | Handle      of handler
  | DoOperation of Name.t * value list * Types.t

type value =
  | Constant    of Constant.t
  | Variable    of var
  | Extend      of value name_map * value option
  | Project     of Name.t * value
  | Erase       of name_set * value
  | Inject      of Name.t * value * Types.t
  | TAbs        of tyvar list * value
  | TApp        of value * tyarg list
  | XmlNode     of Name.t * value name_map * value list
  | ApplyPure   of value * value list
  | Closure     of var * tyarg list * value
  | Coerce      of value * Types.t
```

A.2. WasmIR syntax

WasmIR type	OCaml type	Arguments	Note
`TInt`	`int typ`	None	
`TBool`	`bool typ`	None	
`TFloat`	`float typ`	None	
`TString`	`string typ`	None	
`TClosed`	`('c * 'a -> 'b) typ`	`'a typ_list`, `'b typ`	Type of callables
`TAbsClosArg`	`abs_closure_content typ`	None	Type of abstract closures
`TClosArg`	`'a closure_content typ`	`'a typ_list`	Type of concrete closures
`TCont`	`'a continuation typ`	`'a typ`	
`TTuple`	`'a list typ`	`'a typ_list`	
`TVariant`	`variant typ`	None	Content is boxed
`TList`	`llist typ`	None	
`TVar`	`unit typ`	None	Type of boxed values
`TSpawnLocation`	`Value.spawn_location typ`	None	
`TProcess`	`process typ`	None	
`TLnil`	`unit typ_list`	None	
`TLcons`	`('a * 'b) typ_list`	`'a typ`, `'b typ_list`	

Expression	Parameter(s)
`EUnreachable`	Type
`EConvertClosure`	Abstract closure variable ID, concrete type
`EIgnore`	Expression and its type
`EConstInt`	Integer value
`EConstBool`	Boolean value
`EConstFloat`	Float value
`EConstString`	String value
`EUnop`	Operation, argument expression
`EBinop`	Operation, argument expressions
`EVariable`	Location and ID
`ETuple`	Expression list and the associated type list
`EExtract`	Tuple expression and the extraction ID
`EVariant`	Tag ID, type and expression of content
`EListNil`	Type
`EListHd`	List expression and type
`EListTl`	Type and list expression
`ECase`	Variant expression, expression type, cases list, optional default case
`EClose`	Function ID, closure content
`ERawClose`	Function ID, abstract closure variable ID
`ESpecialize`	Closed expression, specialization and associated boxing information
`EResume`	Continuation expression, argument expression
`ECallRawHandler`	Handler function ID, continuation expression, continuation argument expression, initial loop values expressions, handler closure expression
`ECallClosed`	Closed expression, arguments expression list
`ECond`	Return type, boolean expression, branches blocks
`EDo`	Effect ID, return type, arguments expressions
`EShallowHandle`	Handlee function ID, closure expressions, default case handler, handlers
`EDeepHandle`	Handlee function ID, handlee closure expressions, handler function ID, handler expressions, initial loop values expressions

B. Appendix: test programs

The "slow" sieve program:

```
typename FilterT = [| Test: Int | Emit: (Process({hear:[|Recv:[Int]|]}), [Int]) |];
sig eratosthenes : (Int) ~> [Int]
fun eratosthenes(n) {
  sig new_proc : () ~> Process({ hear:FilterT })
  fun new_proc() {
    spawn {
      sig loop : (Int, Process({ hear:FilterT })) { hear:FilterT }~> ()
      fun loop(n, sub) {
        receive {
          case Emit(p, acc) -> sub ! Emit(p, n :: acc)
          case Test(k) ->
            if (mod(k, n) == 0) ()
            else sub ! Test(k)
        };
        loop(n, sub)
      }
      receive {
        case Emit(p, acc) -> p ! Recv(reverse(acc))
        case Test(n) -> loop(n, new_proc())
      }
    }
  }

  var root_proc = new_proc();
  fun loop(k) {
    if (k > n) {
      spawnWait {
        root_proc ! Emit(self(), []);
        receive { case Recv(v) -> v }
      }
    }
    else {
      root_proc ! Test(k);
      loop(k + 1)
    }
  }
  loop(2)
}
eratosthenes(20000)
```

It creates an actor per prime number, sending a `Test` message for every number to the first actor, then an `Emit` message to gather the result.

The "fast" sieve program:

```
fun gen(n) {
  fun inner(n, aux) {
    if (n == 0) aux
    else inner(n - 1, true :: aux)
  }
  inner(n, [])
}
fun do_filter(filt, n, k) {
  switch (filt) {
    case [] -> []
    case hd :: tl ->
      if (k == n - 1) false :: do_filter(tl, n, 0)
      else hd :: do_filter(tl, n, k + 1)
  }
}
fun sieve(i, filt, acc) {
  switch (filt) {
    case [] -> reverse(acc)
```

```
    case b :: tl ->
      var acc = if (b) i :: acc else acc;
      var filt = if (b) do_filter(tl, i, 0) else tl;
      var i = i + 1;
      sieve(i, filt, acc)
  }
}
sieve(2, gen(20000 - 1), [])
```

There are no arrays in Links, so a linked list is used instead.

The digits-of-pi program:

```
fun gen(n) {
  if (n > 0) 2000 :: gen(n - 1)
  else [0]
}
fun rev_append(l,o) {
  switch (l) {
    case [] -> o
    case x::xs -> rev_append(xs, x::o)
  }
}
fun split_k(r, k, auxr) {
  if (k < 0) { (auxr, r) }
  else {
    var hd = hd(r);
    var tl = tl(r);
    split_k(tl, k - 1, hd :: auxr)
  }
}
fun do_iter(work, tail, i, d) {
  var d = d + hd(work) * 10000;
  var b = 2 * i - 1;
  var hd = mod(d, b);
  var d = d / b;
  var i = i - 1;
  var work = tl(work);
  var tail = hd :: tail;
  if (i > 0) do_iter(work, tail, i, d * i)
  else (rev_append(work, tail), d)
}
fun main(r, k, c, aux) {
  if (k > 0) {
    var (work, tail) = split_k(r, k, []);
    var (tail, d) = do_iter(work, tail, k, 0);
    var hd = c + d / 10000;
    var c = mod(d, 10000);
    main(tail, k - 14, c, hd :: aux)
  } else {
    reverse(aux)
  }
}
main(gen(2800), 2800, 0, [])
```

Source: https://crypto.stanford.edu/pbc/notes/pi/code.html

C. Appendix: bibliography

[1] Alonzo Church. "A formulation of the simple theory of types". In: *Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[2] Amr Sabry and Matthias Felleisen. "Reasoning about programs in continuation-passing style." In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP '92. San Francisco, California, USA: Association for Computing Machinery, 1992, pp. 288–298. ISBN: 0897914813. DOI: 10.1145/141471.141563. URL: https://doi.org/10.1145/141471.141563.

[3] Sam Lindley and James Cheney. "Row-based effect types for database integration". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 91–102. ISBN: 9781450311205. DOI: 10.1145/2103786.2103798. URL: https://doi.org/10.1145/2103786.2103798.

[4] *The Links Programming Language*. URL: https://links-lang.org/.

[5] *WasmFX: Effect Handlers for WebAssembly*. URL: https://wasmfx.dev/.

[6] *WebAssembly*. URL: https://webassembly.org/.