Modular and Effective Concurrency in Functional Programming

Daniel Hillerström
CDT Pervasive Parallelism
daniel.hillerstrom@ed.ac.uk

Abstract

Parallel and concurrent programming is challenging. However, it is important, as parallel architectures have become ubiquitous. If we are to harness the computational power of parallel architectures, then we need structured and modular programming models.

Modular and efficient task scheduling is important in order to achieve performance by picking the most suitable scheduling strategy in a given situation. However, existing programming models require the programmer to annotate blocks of code with inflexible and non-modular compiler directives. Scheduling decisions are made by a generic run-time without exploiting domain knowledge.

If scheduling strategies and tasks were expressible in the same programming language, then programmers could choose the best scheduling strategy for an application. We believe algebraic effect handlers provide a solution to this problem. Therefore, we propose to implement a compiler for Links that generates efficient run-time implementations of algebraic effect handlers. Then we will use handlers to implement modular and effective scheduling.

1. Introduction

During the past decade parallel architectures have become pervasive, especially due to the recent emergence of multicore smartphones. It is difficult to utilise multiple homogeneous processing cores effectively, and the problem is no easier when we consider heterogeneous processing cores. Scheduling is at the heart of parallel and concurrent programming as it is concerned with when computations are executed or suspended. Run-time scheduling is an important problem as the performance goal of a scheduler can be paramount to the overall performance of a program. For example, system schedulers seek to optimise the job throughput. Resource schedulers attempt to ensure fairness by coordinating accesses to shared resources. Task schedulers seek to reduce execution time by considering factors such as dependencies among tasks. In addition, task schedulers may also seek to exploit special-purpose accelerators in the environment in order to speed up task execution.

Clearly, the performance goals of different schedulers may conflict with each other. Furthermore, a scheduler may employ different *scheduling strategies* to achieve its performance goal. The choice of scheduling strategy depends on the problem domain. For instance, stencil computations may benefit from neighbouring points being scheduled together. However, in most programming models the programmer has little or no say regarding scheduling. The programmer's involvement is often limited to annotating blocks of code with compiler directives. But this approach is not modular, because computations and scheduling strategies are hard-wired together. Consequently, it is nontrivial to change or reuse a particular scheduling strategy. In addition, the compiler is free to ignore the programmer's annotation.

We advocate an approach that decouples the implementation of tasks from scheduling strategies. Concretely, we propose to abstract over the choice of scheduling strategy by adapting the approach taken by Multicore OCaml [8] to use Plotkin and Pretnar's *handlers for algebraic effects* [18] to encode scheduling strategies. Specifically, we plan implement a backend for the functional programming language Links [6], and take advantage of its existing implementation of algebraic effect handlers [9] to encode Links' message-passing concurrency model.

2. Problem definition

There exists many concurrent and parallel programming models that attempt to involve programmer participation in scheduling, e.g. CellSs [3], StarSs [16], StarPU [1], OpenMP [15] Chapel [7] and X10 [20] to mention a few established ones.[1] These programming models operate on task graph based descriptions of programs. They map individual tasks onto computational devices automatically.

[1] These programming models are further discussed in Section 6.

However, the disadvantage is, that they provide weak abstractions making scheduling non-modular. The programmer has to resort to compile-time meta programming in a foreign language such as `#pragmas` to influence scheduling. As a consequence the implementation of scheduling strategies and tasks become tightly coupled which makes it nontrivial to swap scheduling strategies. Moreover, if one combines these programming models; what are the semantics of the resulting programming model then?

If we were able to express scheduling strategies and tasks independently of each other, but in the same programming language, then we would be able to give precise semantics. In particular, we would achieve modularity naturally. Therefore, we ask the following question:

How may we reify scheduling strategies as an integrated construct in the host language?

In Section 4 we propose an answer to the question.

3. Background

In Section 3.1 we introduce algebraic effects and handlers, while in Section 3.2 we introduce an implementation of them in the functional language Links.

3.1 Algebraic effects and their handlers

Plotkin and Power introduced algebraic effects [17] for modelling the semantics of effectful computations [11]. An algebraic effect is a collection of abstract operations. For example, we might define an algebraic effect for state: $State(s) \overset{\text{def}}{=} \{Get : s, Put : s \to ()\}$. State is an effect with two operations: Get and Put which retrieve and modify some state s, respectively. The State effect and its operations do not have predefined semantics.

Handlers assign semantics to effects. Essentially, handlers interpret invocations of abstract operations in computations. Handlers may be either closed or open. A closed handler handles a fixed set of effects, whilst an open handler handles a subset of effects. Open handlers can be composed together to handle a larger set of effects. In addition, handlers comes in two different flavours: *deep* or *shallow*. Deep handlers interpret operations uniformly, e.g. every occurrence of Get is interpreted by the same handler. Conversely, shallow handlers interpret operations non-uniformly as operations are not necessarily interpreted by the same handler every time. Both types of handlers have practical applications [10].

3.2 Links with effect handlers

Links is functional programming language [6] with a strong type and effect system. Notably, Links implements both deep and shallow effect handlers as first-class citizens [9]. Intuitively, handlers are interpreters that pattern matches on abstract operations occurring in some handled computation. This is directly reflected in the syntax for handlers. As an example consider the following handler which interprets the State effect introduced in the previous section:

sig state : $\left(() \xrightarrow{\{Get:s, Put:s \to ()\}} a\right) \to (s) \to a$

```
handler state(m)(s) {
  case Get(k)    → k(s)(s)
  case Put(p,k)  → k(())(p)
  case Return(x) → x
}
```

The signature for `state` conveys that it handles some computation `m` whose effect signature contains (only) `Get` and `Put`, the computation returns a value of type `a`. The second parameter `s` is the program state which the computation `m` may modify. The case-statements pattern matches on the operation names in effect signature of `m`. The cases `Get` and `Put` matches on user-defined operations, while the `Return` operation is a special, built-in operation that is invoked implicitly, when the computation `m` returns.

When an operation is invoked in `m` the run-time generates a continuation from that point in the program. Then the run-time transfers control to the handler which exposes this continuation is to the programmer via the parameter `k` in the case-statements. For example, when `Get` is invoked this particular handler invokes the continuation with the current state `s`. This effectively resumes the computation `m` with the state `s`. Moreover, the same state `s` is forwarded to the next invocation of the handler. In the case of `Put`, the handler invokes the continuation with unit, and forwards the new state `p` to subsequent invocations of the handler.

At the time of writing effect handlers are only supported by the Links interpreter. Thus the performance of handlers is low.

4. Proposed solution

We advocate an approach along the lines of Multicore OCaml to use algebraic effect handlers to encode concurrency directly in the host language. The key observation is that concurrent and parallel concepts such as task forking, kernel launch and data-sharing can be thought of as effectful computations. Thus concurrency is encodable through algebraic effects [2, 8]. Handlers will be used to implement scheduling strategies. In Sections 4.1 and 4.2 we give a concrete example of how to encode asynchronous tasks using algebraic effects.

4.1 Encoding cooperative multitasking

We may encode *cooperative multitasking* using the following two operations:

- fork : $(() \to a) \to Fa$
- yield : $()$

The first operation *fork* takes a task as input, and returns a future which eventually will contain the result of the said task. The second operation *yield* suspends the task in which it was invoked. Using these operations we can easily define an asynchronous computation that computes the nth Fibonacci number, e.g.

```
fun fib(n) {
  if (n ≤ 1) {n}
  else {
    var f1 = fork(fun() { fib(n-1) });
    var f2 = fork(fun() { fib(n-2) });
    getf(f1) + getf(f2)
  }
}
```

where `getf` attempts to retrieve the value inside a future. Its implementation is omitted here, but it is simple to implement in terms of yield.[2]

4.2 Scheduling tasks

Next, we are going to implement two schedulers, for the Fibonacci computation, as handlers. First, we implement a scheduler that explores the computation tree in a breath-first manner:

```
open handler sched(m)(f) {
  case Fork(t,k) → {
   var new_f = new_future();
   enqueue(fun(_){sched(t)(Just(new_f))()});
   k(new_f)(f)
  }
  case Yield(k)  → {
   enqueue(fun(_) { k(())(f) });
   var t = dequeue(); t(())
  }
  case Return(x) → {
     putf(f,x);
     if (is_empty()) { x }
     else { var t = dequeue(); t(()) }
   }
}
```

The handler `sched` implements a scheduler with a particular scheduling strategy by pattern matching on occurrences of operations `Fork` and `Yield` in some computation `m`. The handler is parameterised by a future `f`. For instance, when `fork` occurs in `fib` the scheduler first allocates a new future `new_f`. Then it recursively schedules the forked task `t` to run later. Finally, it resumes the task in which `t` was forked by invoking the continuation `k` with the new future `new_f`, the previous future `f` is saved for the next invocation of the scheduler. Observe, that on a single-core platform this scheduling strategy effectively defers execution of forked tasks.

Upon an occurrence of `yield` in `m` the handler suspends the task and schedules it to run later. Thereafter it dequeues and resumes another task `t`.

[2] The full source code is available at `https://raw.githubusercontent.com/dhil/mscr-dissertation/proposal/proposal/coroutines.links`

Finally, when a task finishes the handler puts the return value inside the future `f`. Furthermore, if the task queue is nonempty it dequeues and resumes another task, otherwise it the result of latest task is returned.

This scheduling strategy is suboptimal for the Fibonacci computation on single-core platform, because it unnecessarily materialises the entire computation tree of `fib`(n). Figure 1a depicts the computation tree for `fib`(3). The labels f_i indicate tasks, and the subscripts i their respective global id.

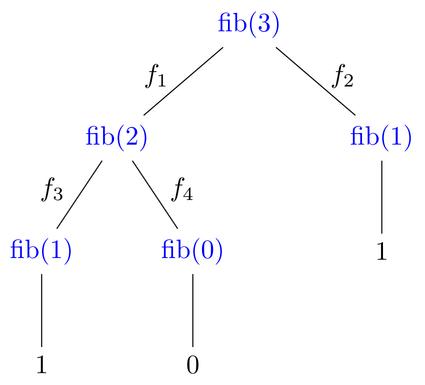
The scheduler fails to exploit the fact that a parent task depends on its two children tasks. We can remedy this issue by eagerly execute forked tasks. Visually, this policy to correspond to visiting the computation tree in a depth-first manner. Such a scheduler is similar to the `sched`, we only need to change queuing policy in the `Fork`-case, e.g.

```
case Fork(t,k) → {
 var new_f = new_future();
 enqueue(fun(_) { k(new_f)(f) });
 sched(t)(Just(new_f))()
}
```

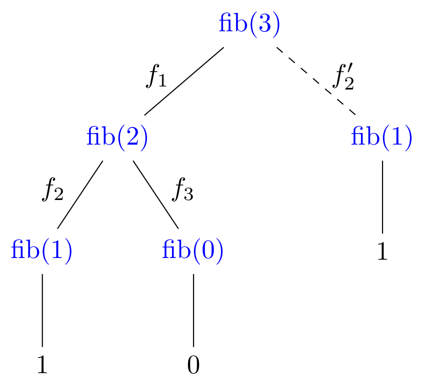
Instead of enqueuing the forked task, we enqueue the continuation to run later. Moreover, we eagerly execute the forked task `t`. Figure 1b depicts the labelled computation tree for `fib(3)` when using this scheduler. Now, the entire computation tree no longer needs to materialised. The space used for tasks in the left subtree can be reused for subsequent tasks in the right subtree. Obviously, we save space, but it has a further subtle impact: The tasks yield fewer times which implies the scheduler is invoked fewer times. Thus, we reduced the overhead incurred by the scheduler. In the case of `fib`(n) it means we, presumably, achieve a small improvement in performance.

Admittedly, the Fibonacci computation is trivial, and we are unlikely to encounter such a trivial computation in any practical application. However, the take away point from this example is, that, we were able change schedulers independently of the computation. Furthermore, we envision that it might be possible to implement scheduler that employ memorisation to optimise computations with repetitive tasks such as the naïve Fibonacci.

Essentially, we derived the sequential schedule for the naïve Fibonacci implementation. If we were to execute the computation on multicore platform, then the breadth-first scheduler would be a better choice. Fortunately, with this programming abstraction it is easy to change schedulers as computations and schedulers are expressed in the same language. In other words: We gain modularity. But this simple exercise is nontrivial in established programming models.



(a) Breadth-first scheduling.



(b) Depth-first scheduling. Note f_2 and f_2' are disjoint tasks.

Figure 1: Task labelled computation trees for `fib(3)`.

4.3 Proposal

We propose to give implement a compiler for the Links programming language that produces efficient run-time implementations of effect handlers. In addition, we hypothesise that the handler abstraction is strong enough for us to achieve modular concurrency. By modular concurrency, we mean that we can change the actual scheduling strategy for processes or threads. Furthermore, we believe that we can encode Links' message-passing concurrency model using handlers.

4.4 Project outcomes

There will be at least two outcomes from this project:

- An efficient implementation of effect handlers.
- An account of concurrent programming with effect encoded concurrency.

5. Methodology

The power of handlers comes from the continuation. At the same time, it is the greatest performance killer, since a continuation is a copy of current stack state. However, one can observe that in many cases continuations are only invoked once, these are called single-shot continuations. A single-shot continuation can be implemented efficiently [5]. But if we restrict all continuations to be single-shot, then the programming model loses a lot of power. Instead we can classify handlers according to how many times they invoke continuations:

0 times *Exception handlers*; no need for continuations.

1 time *Linear handlers*; can be implemented efficiently.

More than 1 time *Multi-handlers*; no known general efficient implementation.

We can take advantage of Links' type system to track this information, then in the code generation phase we may use this information to specialise implementations of particular handlers[3].

6. Related work

In section we briefly summarise related work.

6.1 Concurrent and parallel programming models

Several programming languages attempt to address the problem of scheduling in high-performance computing. A selection of these programming models is listed in Table 1.

CellSs [3] provide a source-to-source compiler that translates annotated C or Fortran code. It attempts to automatically derive opportunities for task-based concurrency. StarSs [16] and StarPU [1] are influenced by CellSs. The former extends CellSs with hierarchical task scheduling. StarPU provides a unified platform for dynamic scheduling on heterogeneous multicore architectures. All three has influenced the open industry standard OpenMP [15].

PARSeC [4] is a generic architecture-aware framework for task scheduling in heterogeneous environments. The run-time operate on task dependency graphs, thus the run-time engine is aware of both the tasks to be executed and the dataflow which connects them. The dataflow representation enables the run-time to handle communication between processes automatically. The run-time inserts synchronisation points whenever necessary.

OpenACC [14] is an open standard for accelerator programming. It uses a combination of programmer annotations via `#pragmas` and APIs in much the same fashion as OpenMP. However, OpenACC focuses exclusively on data parallelism, whereas OpenMP is much broader in its focus.

[3] This idea originated in a discussion with Sam Lindley.

Language	Programming model	Memory model
MPI [12]	Message-passing	Distributed memory
PaRSEC [4]	Dataflow programming	Shared memory & distributed memory
OpenACC [14]	Annotations via pragmas; Data parallel	Shared memory
CellSs [3], StarSs [16], StarPU [1], OpenMP [15]	Annotations via pragmas; API	Shared memory
Chapel [7], X10 [20], Unified Parallel C [19], Coarray Fortran [13]	Shared variable programming	Partitioned Global Address Space

Table 1: Classification of a selection of concurrent and parallel programming models.

MPI is a message-passing model for distributed programming [12]. Programs in MPI are written in Single Program, Multiple Data (SPMD) style. That means all processes execute the same program, but each process may follow designated parts of the program. Data sharing among processes happens via message passing. Communication is two-sided as processes must actively participate in the data-sharing either by sending or receiving.

Chapel [7], Coarray Fortran [13], X10 [20] and Unified Parallel C [19] are so-called Partitioned Global Address Space (PGAS) languages. This programming model provide a shared memory programming interface to distributed programming. All communication between processes are single-sided, i.e. a process can directly manipulate the memory of some remote process.

6.2 Multicore OCaml

OCaml is a functional language with an industrial strength compiler. The Multicore OCaml project [8] attempts to bring multi-core capabilities to the OCaml language by using deep handlers to encode concurrency. The implementation of handlers is restricted to so-called *linear handlers*, i.e. handlers which only invokes the continuation parameter once. However, this restriction is not enforced statically, therefore multiple invocations of a continuation result in a run-time error. Moreover, OCaml does not have an effect system.

7. Evaluation

The primary metric for evaluation of the compiler will be performance of programs produced by it. We intend to mainly measure performance by raw execution speed. Furthermore, we intend to compare our performance results against results for similar programs written in C since it is considered a "performant" language. We will measure the performance on standard multicore desktop machine (Intel i5 CPU).

Additionally, we will consider the safety of our solution, i.e. which static guarantees, if any, can we exhibit. In order to evaluate safety we intend to give a formal proof of the said properties.

Furthermore, we would like to qualitatively assess the ease of use, that is, whether the handler abstraction makes concurrent programming easier.

8. Project plan

Activity / Weeks	2-4	5-9	10-18	19-21	22-33
Finding examples	X	X			
Related work	X	X			X
Benchmarks	X	X	X		
Prototyping	X	X			
Implementation		X	X	X	X
Testing		X	X		X
Single-core eval.			X		
Multi-core eval.				X	X
Safety eval.					X
Write up	X	X	X	X	X

Table 2: Coarse-grained activity overview.

Table 2 provide a high-level overview for the activities during this project. The first three activities are connected. The idea is to survey related and background material to find examples and applications that we can use to build a benchmark suite. Furthermore, we intend to revisit the related work activity during later phases of the project in order to potentially include and discuss the latest work in the dissertation.

The prototyping and implementation activities are connected aswell. We are going to create some prototype implementations based on the examples and applications that we found. These prototypes will possibly be use-case specific, however, the idea is to use them as a starting point for the implementation of the general solution. The implementation activity is going to be the main activity which is reflected in the table. The activities implementation and testing could arguably have been combined into one activity, however, here we have separated them to emphasise the importance of both activities. The idea is to adopt an agile development approach by decomposing the implementation activity into many small tasks. The duration of a single task should be at most one week. Furthermore, we will have automated regression testing that tests the correctness of the implementation. This ought to ensure that sensible progress is made (at least up to the test coverage).

Evaluation will become a natural extension of the testing activity. We split evaluation into three activities: Single-core, multi-core and safety. First we will evaluate the performance of programs that use a single core. Test programs will include both concurrent and sequential programs. Later, this activity will be subsumed by the multi-core activity in which we will consider parallel execution of programs. During the evaluation activities we shift the focus from correctness to fulfilment of our goals. The implementation will be evaluated weekly with regard to the performance goals. Possibly, this will be automated to a certain extent. Evaluation of safety and ease of use will be done manually. The former is a pencil and paper exercise, while the latter requires a subjective comparison of code examples.

The dissertation write up is supposed to be a continuous activity. The aim is to produce a work sheet per week that reflects the work done in the other activities. They will act as a sort of worklog. Moreover, the worksheets are not necessarily meant to become part of the dissertation. But as we approach the hand-in deadline, we will shift more focus onto writing up the actual dissertation. Hopefully, the worksheets can provide the foundation for the dissertation.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logic and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[3] P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. Cellss: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 15(6):36–45, Nov. 2013.

[5] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. *SIGPLAN Not.*, 31(5):99–107, May 1996.

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[7] Cray Inc. Chapel language specifiction version 0.91, 2012.

[8] S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects, 9 2015. OCaml Workshop.

[9] D. Hillerström. Handlers for algebraic effects in links. Master's thesis, School of Informatics, the University of Edinburgh, Scotland, 2015. Supervised by Sam Lindley.

[10] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

[11] S. Lindley. Algebraic effects and effect handlers for idioms and arrows. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 47–58. ACM, 2014.

[12] Message Passing Interface Forum. MPI: A message-passing interface standard version 3.0, 2012.

[13] R. W. Numrich and J. Reid. Co-arrays in the next Fortran Standard, 2005.

[14] OpenACC-Standard.org. OpenACC application programming interface version 2.0, 2013.

[15] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[16] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *IJHPCA*, 23(3):284–299, 2009.

[17] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[18] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[19] UPC Consortium. UPC language specifications version 1.3, 2013.

[20] I. P. O. T. Vijay Saraswat, Bard Bloom and D. Grove. X10 language specification, 2015.