Under consideration for publication in J. Functional Programming

Arrows for Parallel Computations

MARTIN BRAUN
University Bayreuth, 95440 Bayreuth, Germany
PHIL TRINDER
Glasgow University, Glasgow, G12 8QQ, Scotland
and OLEG LOBACHEV
University Bayreuth, 95440 Bayreuth, Germany

Abstract

Arrows are a general interface for computation and therefore form an alternative to monads for API design. We express parallelism using this concept in a novel way: We define an arrows-based language for parallelism and implement it using multiple parallel Haskells. In this manner we are able to bridge across various parallel Haskells.

Additionally, our way of writing parallel programs has the benefit of being portable across flavours of parallel Haskells. Furthermore, as each parallel computation is an arrow, which means that they can be composed and transformed as such. We introduce some syntactic sugar to provide parallelism-aware arrow combinators.

To show that our arrow-based language is on par with the existing parallel languages, we also define several parallel skeletons with our framework. Benchmarks show that our framework does not induce too much overhead performance-wise.

Contents

Contents

1 Introduction

OL: todo, reuse 5.5, and more

blablabla arrows, parallel, haskell.

Contribution **OL: HIT HERE REALLY STRONG**

MB: different, how? We wrap parallel Haskells inside of our *ArrowParallel* interface, but why do we aim to abstract parallelism this way and what does this approach do better than the other parallel Haskells?

- **Arrow DSL benefits**: With the *ArrowParallel* typeclass we do not lose any benefits of using arrows as *parEvalN* is just yet another arrow combinator. The resulting arrow can be used in the same way a potential serial version could be used. This is a big advantage of this approach, especially compared to the monad solutions as we do not introduce any new types. We can just ‘plug’ in parallel parts into sequential arrow-based programs without having to change anything.
- **Abstraction**: With the *ArrowParallel* typeclass, we abstract all parallel implementation logic away from the business logic. This leaves us in the beautiful situation of being able to write our code against the interface the typeclass gives us without being bound to any parallel Haskell. So as an example, during development, we can run the program in a simple GHC-compiled variant and afterwards deploy it on a cluster by converting it into an Eden version, by just replacing the actual *ArrowParallel* instance.

Structure The remaining text is structures as follows. Section 2 briefly introduces known parallel Haskell flavours and gives an overview of Arrows to the reader (Sec. 2.2). Section 3

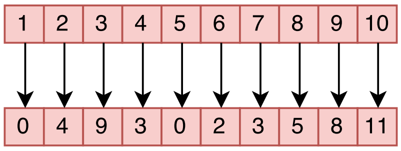


Figure 1: Schematic illustration of *parEvalN*.

discusses related work. Section 4 defines Parallel Arrows and presents a basic interface. Section 5 defines Futures for Parallel Arrows, this concept enables better communication. Section 6 presents some basic algorithmic skeletons (parallel *map* with and without load balancing, *map* − *reduce*) in our newly defined dialect. More advanced ones are showcased in Section 7 (*pipe*, *ring*, *torus*). Section 8 shows the benchmark results. Section 9 discusses future work and concludes.

2 Background

2.1 Short introduction to parallel Haskells

There are already several ways to write parallel programs in Haskell. As we will base our parallel arrows on existing parallel Haskells, we will now give a short introduction to the ones we use as backends in this paper.

In its purest form, parallel computation (on functions) can be looked at as the execution of some functions $a \rightarrow b$ in parallel or $\mathit{parEvalN} :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$, as also Figure 1 symbolically shows. Before we go into detail on how we can use this idea of parallelism for parallel Arrows, as a short introduction to parallelism in Haskell we will now implement *parEvalN* with several different parallel Haskells.

2.1.1 Multicore Haskell

Multicore Haskell (Marlow *et al.*, 2009; Trinder *et al.*, 1998) is way to do parallel processing found in standard GHC.[1] It ships with parallel evaluation strategies for several types which can be applied with $\mathit{using} :: a \rightarrow \mathit{Strategy}\ a \rightarrow a$.

For *parEvalN* this means that we can just apply the list of functions $[a \rightarrow b]$ to the list of inputs $[a]$ by zipping them with the application operator $. We then evaluate this lazy list $[b]$ according to a $\mathit{Strategy}\ [b]$ with the $\mathit{using} :: a \rightarrow \mathit{Strategy}\ a \rightarrow a$ operator. We construct this strategy with $\mathit{parList} :: \mathit{Strategy}\ a \rightarrow \mathit{Strategy}\ [a]$ and $\mathit{rdeepseq} :: \mathit{NFData}\ a \Rightarrow \mathit{Strategy}\ a$ where the latter is a strategy which evalutes to normal form. To ensure that programs that use *parEvalN* have the correct evaluation order, we annotate the computation with $\mathit{pseq} :: a \rightarrow b \rightarrow b$ which forces the compiler to not reorder multiple *parEvalN* computations. This is particularly necessary in circular communication topologies like in the *torus* or *ring* (Chap. 7), where a wrong execution order would result in deadlock scenarios when executed without *pseq*.

[1] Multicore Haskell on Hackage is available under `https://hackage.haskell.org/package/parallel-3.2.1.0`, compiler support is integrated in the stock GHC.

The resulting code and a graphical representation can be found in Fig. 2 and Fig. 3, respectively.

$$
\begin{aligned}
&\mathit{parEvalN} :: (\mathit{NFData}\ b) \Rightarrow [a \to b] \to [a] \to [b] \\
&\mathit{parEvalN}\ \mathit{fs}\ \mathit{as} = \mathbf{let}\ \mathit{bs} = \mathit{zipWith}\ (\$)\ \mathit{fs}\ \mathit{as} \\
&\quad \mathbf{in}\ (\mathit{bs}\ `\mathit{using}`\ \mathit{parList}\ \mathit{rdeepseq})\ `\mathit{pseq}`\ \mathit{bs}
\end{aligned}
$$

Figure 2: Multicore version of *parEvalN*.

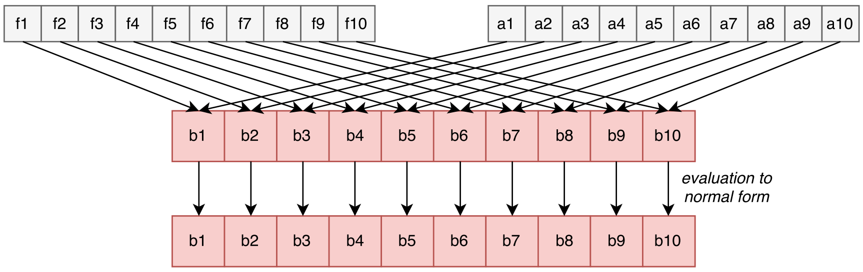


Figure 3: Dataflow of the Multicore Haskell *parEvalN* version.

2.1.2 ParMonad

The *Par* monad[2] introduced by Marlow *et al.* (2011a), is a monad designed for composition of parallel programs.

The *Par* Monad version of our parallel evaluation function *parEvalN* can be defined by zipping the list of $[a \to b]$ with the list of inputs $[a]$ with the application operator \$ just like with Multicore Haskell. Then, we map over this not yet evaluated lazy list of results $[b]$ with $\mathit{spawnP} :: \mathit{NFData}\ a \Rightarrow a \to \mathit{Par}\ (\mathit{IVar}\ a)$ to transform them to a list of not yet evaluated forked away computations $[\mathit{Par}\ (\mathit{IVar}\ b)]$, which we convert to $\mathit{Par}\ [\mathit{IVar}\ b]$ with *sequenceA*. We wait for the computations to finish by mapping over the *IVar b* values inside the *Par* monad with *get*. This results in $\mathit{Par}\ [b]$. We execute this process with *runPar* to finally get the final $[b]$.

Again, the resulting code and a graphical representation can be found in Fig. 4 and Fig. 5, respectively.

MB: explain problems with laziness here. Problems with torus

2.1.3 Eden

Eden (Loogen *et al.*, 2005; Loogen, 2012) is a parallel Haskell for distributed memory and comes with a MPI and a PVM backends.[3] It is targeted towards clusters, but also functions

[2] It can be found in the `monad-par` package on hackage under `https://hackage.haskell.org/package/monad-par-0.3.4.8/`.

[3] See also `http://www.mathematik.uni-marburg.de/~eden/` and `https://hackage.haskell.org/package/edenmodules-1.2.0.0/`.

$$\begin{aligned}&\mathit{parEvalN} :: (\mathit{NFData}\ b) \Rightarrow [a \to b] \to [a] \to [b]\\&\mathit{parEvalN}\ \mathit{fs}\ \mathit{as} = \mathit{runPar}\ \$\\&\quad (\mathit{sequenceA}\ \$\ \mathit{map}\ (\mathit{spawnP})\ \$\ \mathit{zipWith}\ (\$)\ \mathit{fs}\ \mathit{as}) \gg\!= \mathit{mapM}\ \mathit{get}\end{aligned}$$

Figure 4: *Par* Monad version of *parEvalN*.

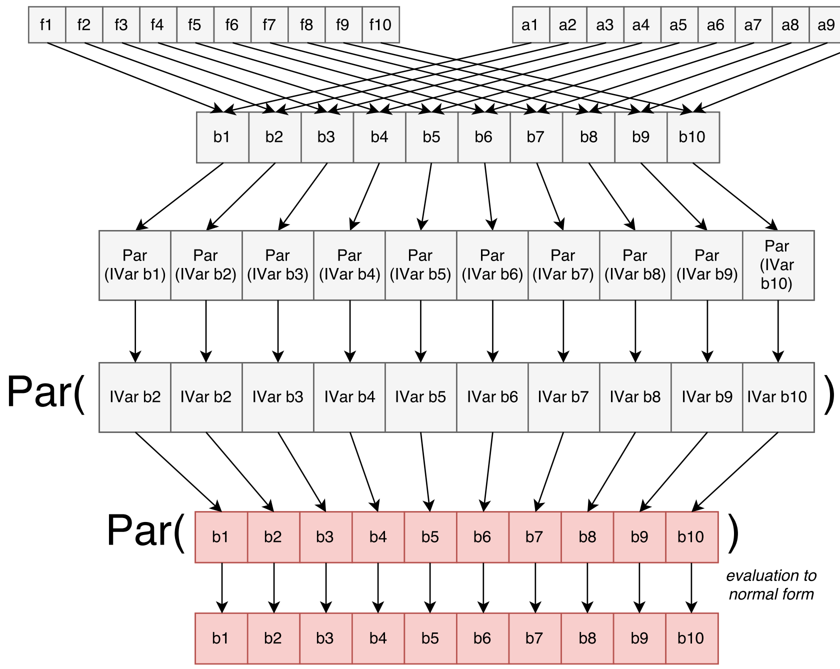


Figure 5: Dataflow of the *Par* Monad *parEvalN* version.

well in a shared-memory setting with a further simple backend. However, in contrast to many other parallel Haskells, in Eden each process has its own heap. This seems to be a waste of memory, but with distributed programming paradigm and individual GC per process, Eden yields good performance results also on multicores (Berthold *et al.*, 2009a; Aswad *et al.*, 2009).

While Eden also comes with a monad *PA* for parallel evaluation, it also ships with a completely functional interface that includes a $\mathit{spawnF} :: (\mathit{Trans}\ a, \mathit{Trans}\ b) \Rightarrow [a \to b] \to [a] \to [b]$ function that allows us to define *parEvalN* directly:

$$\begin{aligned}&\mathit{parEvalN} :: (\mathit{Trans}\ a, \mathit{Trans}\ b) \Rightarrow [a \to b] \to [a] \to [b]\\&\mathit{parEvalN} = \mathit{spawnF}\end{aligned}$$

A simplistic graphical depiction of this definition can be found in Fig. 6.

Eden TraceViewer. To comprehend the efficiency and the lack thereof in a parallel program, an inspection of its execution is extremely helpful. While some large-scale solutions exist (Geimer *et al.*, 2010), the parallel Haskell community mainly utilises the

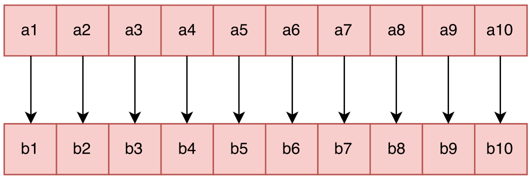


Figure 6: Dataflow of the Eden *parEvalN* version.

tools Threadscope (Wheeler & Thain, 2009) and Eden TraceViewer[4] (Berthold & Loogen, 2007). In the next sections we will present some *traces*, the post-mortem process diagrams of Eden processes and their activity.

In a trace, the x axis shows the time, the y axis enumerates the machines and processes. A trace shows a running process in green, a blocked process is red. If the process is 'runnable', i.e. it may run, but does not, it is yellow. The typical reason for then is GC. An inactive machine where no processes are started yet, or all are already terminated, is shows as a blue bar. A comminication from one process to another is represented with a black arrow. A stream of communications, e.g. a transmitted list is shows as a dark shading between sender and receiver processes. An example trace can be found in Fig. 18.

2.2 Arrows

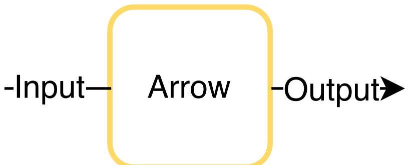


Figure 7: Schematic depiction of an arrow.

Arrows were introduced by Hughes (2000) as a general interface for computation. An arrow *arr a b* represents a computation that converts an input *a* to an output *b*. This is defined in the *Arrow* shown in Fig. 8.

Its *arr* operation is used to lift an ordinary function to the specified arrow type, similarly to the monadic *return*. The $>>>$ operator is analogous to the monadic composition $\gg=$ and combines two arrows *arr a b* and *arr b c* by "wiring" the outputs of the first to the inputs to the second to get a new arrow *arr a c*. Lastly, the *first* operator takes the input arrow from *b* to *c* and converts it into an arrow on pairs with the second argument untouched. It allows us to to save input across arrows.

The most prominent instances of this interface are regular functions $(\rightarrow)$:

[4] See `http://hackage.haskell.org/package/edentv` on Hackage for the last available version of Eden TraceViewer.

class *Arrow arr* **where**
  $arr :: (a \to b) \to arr\ a\ b$
  $(>>>) :: arr\ a\ b \to arr\ b\ c \to arr\ a\ c$
  $first :: arr\ a\ b \to arr\ (a,c)\ (b,c)$

Figure 8: *Arrow* class definition.

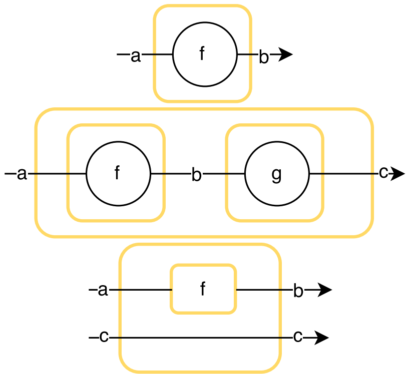


Figure 9: schematic depiction of *Arrow* combinators *arr*, $>>>$ and *first*.

instance *Arrow* $(\to)$ **where**
  $arr\ f = f$
  $f >>> g = g \circ f$
  $first\ f = \lambda(a,c) \to (f\ a, c)$

and the Kleisli type:

data $Kleisli\ m\ a\ b = Kleisli\ \{run :: a \to m\ b\}$
instance $Monad\ m \Rightarrow Arrow\ (Kleisli\ m)$ **where**
  $arr\ f = Kleisli\ (return \circ f)$
  $f >>> g = Kleisli\ (\lambda a \to f\ a \gg= g)$
  $first\ f = Kleisli\ (\lambda(a,c) \to f\ a \gg= \lambda b \to return\ (b,c))$

With this typeclass in place, Hughes also defined some syntactic sugar (Fig. 10): The

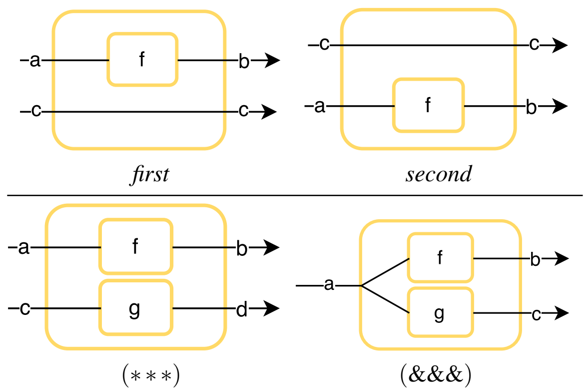


Figure 10: Visual depiction of syntactic sugar for arrows.

combinators *second*, $***$ and &&&. The definition of *second*, the mirrored version of *first*, is:

$$\begin{array}{l} \mathit{second} :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \rightarrow \mathit{arr}\ (c,a)\ (c,b) \\ \mathit{second}\ f = \mathit{arr}\ \mathit{swap} >>> \mathit{first}\ f >>> \mathit{arr}\ \mathit{swap} \\ \quad \textbf{where}\ \mathit{swap}\ (x,y) = (y,x) \end{array}$$

The $***$ combinator that combines *first* and *second* to handle two inputs in one arrow, is defined as

$$\begin{array}{l} (***) :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \rightarrow \mathit{arr}\ c\ d \rightarrow \mathit{arr}\ (a,c)\ (b,d) \\ f *** g = \mathit{first}\ f >>> \mathit{second}\ g \end{array}$$

, while the &&& combinator, that constructs an arrow which outputs two different values like $***$, but takes only one input, is:

$$\begin{array}{l} (\&\&\&) :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \rightarrow \mathit{arr}\ a\ c \rightarrow a\ a\ (b,c) \\ f\ \&\&\&\ g = \mathit{arr}\ (\lambda a \rightarrow (a,a)) >>> (f *** g) \end{array}$$

A first short example given by Hughes on how to use arrows is addition with arrows:

$$\begin{array}{l} \mathit{add} :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ \mathit{Int} \rightarrow \mathit{arr}\ a\ \mathit{Int} \rightarrow \mathit{arr}\ a\ \mathit{Int} \\ \mathit{add}\ f\ g = (f\ \&\&\&\ g) >>> \mathit{arr}\ (\lambda(u,v) \rightarrow u+v) \end{array}$$

The more restrictive interface of arrows (a monad can be *anything*, an arrow is a process of doing something, a *computation*) allows for more elaborate composition and transformation combinators. One of the major problems in parallel computing is composition of parallel processes.

3 Related Work

OL: arrows or Arrows?

3.1 Parallel Haskells

Of course, the three parallel Haskell flavours we have presented above: the GpH (Trinder *et al.*, 1996, 1998) parallel Haskell dialect and its multicore version (Marlow *et al.*, 2009), the *Par* monad (Marlow *et al.*, 2011b; Foltzer *et al.*, 2012), and Eden (Loogen *et al.*, 2005; Loogen, 2012) are related to this work. We use these languages as backends: our DSL can switch from one to other at user's command.

HdpH (Maier *et al.*, 2014; Stewart *et al.*, 2016) is an extension of *Par* monad to heterogeneous clusters. LVish (Kuper *et al.*, 2014) is a communication-centred extension of *Par* monad. Further parallel Haskell approaches include pH (Nikhil & Arvind, 2001), research work done on distributed variants of GpH (Trinder *et al.*, 1996; Aljabri *et al.*, 2014, 2015), and low-level Eden implementation (Berthold, 2008; Berthold *et al.*, 2016). Skeleton composition (Dieterle *et al.*, 2016), communication (Dieterle *et al.*, 2010a), and generation of process networks (Horstmeyer & Loogen, 2013) are recent in-focus research topics in Eden. This also includes the definitions of new skeletons (Hammond *et al.*, 2003; Berthold & Loogen, 2006; Berthold *et al.*, 2009b,c; Dieterle *et al.*, 2010b; de la Encina *et al.*, 2011; Dieterle *et al.*, 2013; Janjic *et al.*, 2013).

More different approaches include data parallelism (Chakravarty *et al.*, 2007; Keller *et al.*, 2010), GPU-based approaches (Mainland & Morrisett, 2010; Svensson, 2011), software transactional memory (Harris *et al.*, 2005; Perfumo *et al.*, 2008). The Haskell–GPU bridge Accelerate (Chakravarty *et al.*, 2011; Clifton-Everest *et al.*, 2014; McDonell *et al.*, 2015) deserves a special mention. Accelerate is completely orthogonal to our approach. Marlow authored a recent book in 2013 on parallel Haskells.

3.2 Algorithmic skeletons

Algorithmic skeletons were introduced by Cole (1989). Early efforts include (Darlington *et al.*, 1993; Botorog & Kuchen, 1996; Danelutto *et al.*, 1997; Gorlatch, 1998; Lengauer *et al.*, 1997). Rabhi & Gorlatch (2003) consolidated early reports on high-level programming approaches. The effort is ongoing, including topological skeletons (Berthold & Loogen, 2006), special-purpose skeletons for computer algebra (Berthold *et al.*, 2009c; Lobachev, 2011, 2012; Janjic *et al.*, 2013), iteration skeletons (Dieterle *et al.*, 2013). The idea of Linton *et al.* (2010) is to use a parallel Haskell to orchestrate further software systems to run in parallel. Dieterle *et al.* (2016) compare the composition of skeletons to stable process networks.

3.3 Arrows

Arrows were introduced by Hughes (2000), basically they are a generalised function arrow $\rightarrow$. Hughes (2005a) is a tutorial on arrows. Some theoretical details on arrows (Jacobs *et al.*, 2009; Lindley *et al.*, 2011; Atkey, 2011) are viable. Paterson (2001) introduced a new notation for arrows. Arrows have applications in information flow research (Li & Zdancewic, 2006, 2010; Russo *et al.*, 2008), invertible programming (Alimarine *et al.*, 2005), and quantum computer simulation (Vizzotto *et al.*, 2006). But perhaps most prominent application of arrows is arrow-based functional reactive programming, AFRP (Hudak *et al.*, 2003).**OL: cite more!**

Liu *et al.* (2009) formally define a more special kind of arrows that capsule the computation more than regular arrows do and thus enable optimizations. Their approach would allow parallel composition, as their special arrows would not interfere with each other in concurrent execution. In contrast, we capture a whole parallel computation as a single entity: our main instantiation function *parEvalN* makes a single (parallel) arrow out of list of arrows.**OL: ugh, take care!** Huang *et al.* (2007) utilise arrows for parallelism, but strikingly different from our approach. They basically use arrows to orchestrate several tasks in robotics. We, however, propose a general interface for parallel programming, remaining completely in Haskell.

3.4 Other languages

Although this work is centred on Haskell implementation of arrows, it is applicable to any functional programming language where parallel evaluation and arrows can be defined. Experiments with our approach in Frege language [5] (which is basically Haskell on the JVM) were quite successful, we were able to use typical Java libraries for parallelism **MB:**

[5] GitHub project page at `https://github.com/Frege/frege`

not really tested completely. basic parEvalN worked, we didn't test for different Java ExecutorServices, though.... However, it is beyond the scope of this work.

Achten *et al.* (2004, 2007) use an arrow implementation in Clean for better handling of typical GUI tasks. Dagand *et al.* (2009) used arrows in OCaml in the implementation of a distributed system.

4 Parallel Arrows

We have seen what Arrows are and how they can be used as a general interface to computation. In the following section we will discuss how Arrows constitute a general interface not only to computation, but to *parallel computation* as well. We start by introducing the interface and explaining the reasonings behind it. Then, we discuss some implementations using exisiting parallel Haskells. Finally, we explain why using Arrows for expressing parallelism is beneficial.

4.1 The ArrowParallel typeclass

As we have seen earlier, in its purest form, parallel computation (on functions) can be seen as the execution of some functions $a \to b$ in parallel, *parEvalN* (Chap. 2.1). Translating this into arrow terms gives us a new operator *parEvalN* that lifts a list of arrows $[arr\ a\ b]$ to a parallel arrow $arr\ [a]\ [b]$. This combinator is similar to our utility function *listApp* from Appendix A, but does parallel instead of serial evaluation.

$$parEvalN :: (Arrow\ arr) \Rightarrow [arr\ a\ b] \to arr\ [a]\ [b]$$

With this definition of *parEvalN*, parallel execution is yet another arrow combinator. But as the implementation may differ depending on the actual type of the arrow *arr* and we want this to be an interface for different backends, we introduce a new typeclass *ArrowParallel arr a b*:

$$\begin{array}{l}\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ \textbf{where}\\ \quad parEvalN :: [arr\ a\ b] \to arr\ [a]\ [b]\end{array}$$

Sometimes parallel Haskells require or allow for additional configuration parameters, e. g. an information about the execution environment or the level of evaluation (weak head normal form vs. normal form). For this reason we also introduce an additional *conf* parameter to the function. We also do not want *conf* to be a fixed type, as the configuration parameters can differ for different instances of *ArrowParallel*. So we add it to the type signature of the typeclass as well: **OL:** *ArrowParallel arr a b conf* **or** *ArrowParallel conf arr a b* **? MB: does it really matter?**

$$\begin{array}{l}\textbf{class}\ Arrow\ arr \Rightarrow ArrowParallel\ arr\ a\ b\ conf\ \textbf{where}\\ \quad parEvalN :: conf \to [arr\ a\ b] \to arr\ [a]\ [b]\end{array}$$

Note that we don't require the *conf* parameter in every implementation. If it is not needed, we usually just default the *conf* type parameter to $()$ and even blank it out in the parameter list of the implemented *parEvalN*, as we will see in the implementation of the Multicore Haskell and the *Par* Monad backends.

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) => ArrowParallel arr a b () where
  parEvalN _ fs =
    listApp fs >>>
    arr (withStrategy (parList rdeepseq)) &&& arr id >>>
    arr (uncurry pseq)
```

Figure 11: Fully evaluating ArrowParallel instance for the Multicore Haskell backend.

4.2 ArrowParallel instances

4.2.1 Multicore Haskell

The Multicore Haskell implementation of *ArrowParallel* is implemented in a straightforward manner by using *listApp* (Appendix A) combined with the $withStrategy :: Strategy\ a \rightarrow a \rightarrow a$ and $pseq :: a \rightarrow b \rightarrow b$ combinators from Multicore Haskell, where *withStrategy* is the same as $using :: a \rightarrow Strategy\ a \rightarrow a$ but with flipped parameters. For most cases a fully evaluating version like in Fig. 11 would probably suffice, but as the Multicore Haskell interface allows the user to specify the level of evaluation to be done via the *Strategy* interface, our DSL should allow for this. We therefore introduce the *Conf a* data-type that simply wraps a *Strategy a*:

```
data Conf a = Conf (Strategy a)
```

We can't directly use the *Strategy a* type here as GHC (at least currently) does not allow type synonyms in type class instances. To get our configurable *ArrowParallel* instance, we simply unwrap the strategy and pass it to *parList* like in the fully evaluating version (Fig. 12).

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
  ArrowParallel arr a b (Conf b) where
  parEvalN (Conf strat) fs =
    listApp fs >>>
    arr (withStrategy (parList strat)) &&& arr id >>>
    arr (uncurry pseq)
```

Figure 12: Configurable ArrowParallel instance for the Multicore Haskell backend.

4.2.2 Par Monad

OL: introduce a newcommand for par-monad, "arrows", "parrows" and replace all mentions to them to ensure uniform typesetting The *Par* monad implementation (Fig. 13) makes use of Haskells laziness and *Par* monad's $spawnP :: NFData\ a \Rightarrow a \rightarrow Par\ (IVar\ a)$ function. The latter forks away the computation of a value and returns an *IVar* containing the result in the *Par* monad.

We therefore apply each function to its corresponding input value with | and then fork the computation away with *arr spawnP* inside a *zipWithArr* (Fig. A 3) call. This yields a list $[Par\ (IVar\ b)]$, which we then convert into $Par\ [IVar\ b]$ with *arr sequenceA*. In order to wait for the computation to finish, we map over the *IVar*s inside the *Par* monad with $arr\ (\gg= mapM\ get)$. The result of this operation is a $Par\ [b]$ from which we can finally remove the monad again by running *arr runPar* to get our output of $[b]$.

```
instance (NFData b, ArrowApply arr, ArrowChoice arr) ⇒
  ArrowParallel arr a b conf where
    parEvalN _ fs =
      (arr $ λas → (fs, as)) >>>
      zipWithArr (app >>> arr spawnP) >>>
      arr sequenceA >>>
      arr (≫= mapM get) >>>
      arr runPar
```

Figure 13: ArrowParallel instance for the Par Monad backend.

4.2.3 Eden

For both the Multicore Haskell and *Par* Monad implementations we could use general instances of *ArrowParallel* that just require the *ArrowApply* and *ArrowChoice* typeclasses. With Eden this is not the case as we can only spawn a list of functions and we cannot extract simple functions out of arrows. While we could still manage to have only one class in the module by introducing a typeclass:

```
class (Arrow arr) ⇒ ArrowUnwrap arr where
  arr a b → (a → b)
```

We don't do it here for aesthetic resons, though. For now, we just implement *ArrowParallel* for normal functions:

```
instance (Trans a, Trans b) ⇒ ArrowParallel (→) a b conf where
parEvalN _ fs as = spawnF fs as
```

and the Kleisli type:

```
instance (Monad m, Trans a, Trans b, Trans (m b)) ⇒
  ArrowParallel (Kleisli m) a b conf where
parEvalN conf fs =
  (arr $ parEvalN conf (map (λ(Kleisli f) → f) fs)) >>>
  (Kleisli $ sequence)
```

4.3 Extending the Interface

With the *ArrowParallel* typeclass in place and implemented, we can now implement some further basic parallel interface functions. These are algorithmic skeletons that, however, mostly serve as a foundation to further, more specific algorithmic skeletons.

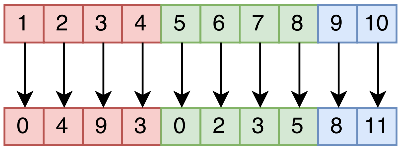


Figure 14: Schematic depiction of *parEvalNLazy*.

```haskell
parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply arr) =>
  conf -> ChunkSize -> [arr a b] -> (arr [a] [b])
parEvalNLazy conf chunkSize fs =
  arr (chunksOf chunkSize) >>>
  listApp fchunks >>>
  arr concat
  where fchunks = map (parEvalN conf) $ chunksOf chunkSize fs
```

Figure 15: Definition of *parEvalNLazy*.

4.3.1 Lazy parEvalN

The function *parEvalN* is 100% strict, which means that it fully evaluates all passed arrows. Sometimes this might not be feasible, as it will not work on infinite lists of functions like e.g. $map\ (arr \circ (+))\ [1\,..]$ or just because we need the arrows evaluated in chunks. *parEvalNLazy* (Fig. 14, 15) fixes this. It works by first chunking the input from $[a]$ to $[[a]]$ with the given *ChunkSize* in *arr* $(chunksOf\ chunkSize)$. These chunks are then fed into a list $[arr\ [a]\ [b]]$ of parallel arrows created by feeding chunks of the passed *ChunkSize* into the regular parEvalN by using *listApp*. The resulting $[[b]]$ is lastly converted into $[b]$ with *arr concat*.

4.3.2 Heterogenous tasks

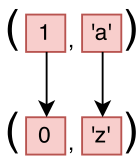


Figure 16: Schematic depiction of *parEval2*.

We have only talked about the paralellization arrows of the same type until now. But sometimes we want to paralellize heterogenous types as well. However, we can implement such a *parEval2* combinator (Fig. 16, B 4) which combines two arrows *arr a b* and *arr c d* into a new parallel arrow $arr\ (a,c)\ (b,d)$ quite easily with the help of the *ArrowChoice* typeclass. The idea is to use the $+\!+\!+$ combinator which combines two arrows *arr a b* and *arr c d* and transforms them into $arr\ (Either\ a\ c)\ (Either\ b\ d)$ to get a common arrow type that we can then feed into parEvalN.

5 Futures

Consider the parallel arrow combinator in Fig. 17 In a distributed environment, a resulting

$$\mathit{someCombinator} :: (\mathit{Arrow}\ \mathit{arr}) \Rightarrow [\mathit{arr}\ a\ b] \rightarrow [\mathit{arr}\ b\ c] \rightarrow \mathit{arr}\ [a]\ [c]$$
$$\mathit{someCombinator}\ \mathit{fs1}\ \mathit{fs2} = \mathit{parEvalN}\ ()\ \mathit{fs1} >>> \mathit{rightRotate} >>> \mathit{parEvalN}\ ()\ \mathit{fs2}$$

Figure 17: An example parallel Arrow combinator without Futures.

arrow of this combinator first evaluates all $[arr\ a\ b]$ in parallel, sends the results back to the master node, rotates the input once and then evaluates the $[arr\ b\ c]$ in parallel to then gather the input once again on the master node. Such situations arise, e.g. in scientific computations when the data distributed across the nodes needs to be transposed. A concrete example is 2D FFT computation (Gorlatch & Bischof, 1998; Berthold *et al.*, 2009c).

While the example in Fig. 17 could be rewritten into only one *parEvalN* call by directly wiring the arrows properly together, this example illustrates an important problem: When using a *ArrowParallel* backend that resides on multiple computers, all communication between the nodes is done via the master node, as shown in the Eden trace in Figure 18. This can become a serious bottleneck for larger amount of data and number of processes (showcases Berthold *et al.*, 2009c, as, e.g.).

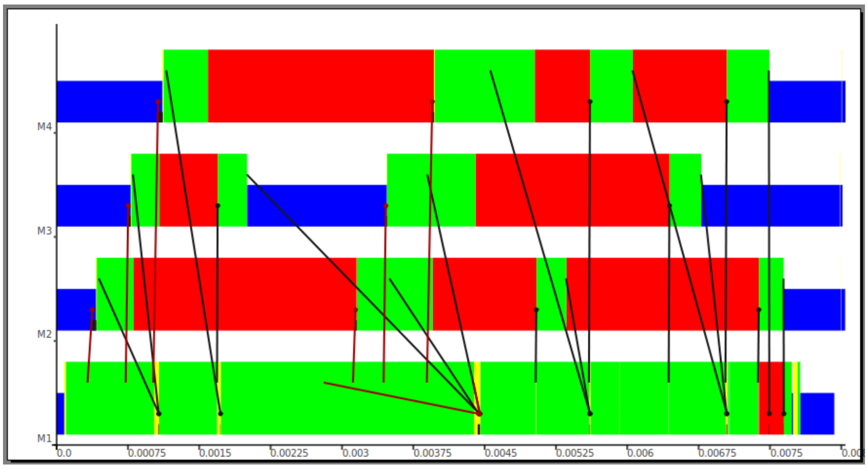


Figure 18: Communication between 4 threads without Futures. All communication goes through the master node. Each bar represents one process. Black lines between processes represent communication. Colors: blue ≙ idle, green ≙ running, red ≙ blocked, yellow ≙ suspended.

OL: more practical and heavy-weight example! fft (I have the code)?
MB: Depends... Are the communications easy to read in such an example?
MB: Keep the description for the different colours, or link to the EdenTV description in 2.1.3

This motivates for an approach that allows the nodes to communicate directly with each other. Thankfully, Eden, the distributed parallel Haskell we have used in this paper so far, already ships with the concept of *RD* (remote data) that enables this behaviour (Alt & Gorlatch, 2003; Dieterle *et al.*, 2010a).

But as we want code written against our API to be implementation agnostic, we have to wrap this context. We do this with the *Future* typeclass (Fig. 19). Since *RD* is only a

```
class Future fut a | a -> fut where
  put :: (Arrow arr) => arr a (fut a)
  get :: (Arrow arr) => arr (fut a) a
```

Figure 19: Definition of the *Future* typeclass.

type synonym for communication type that Eden uses internally, we have to use some wrapper classes to fit that definition, though, as seen in Appendix in Fig. B 1. This is due to the same reason we had to introduce a wrapper for *Strategy a* in the Multicore Haskell implementation of *ArrowParallel* in Section 4.2.1.

For our Par Monad and Multicore Haskell backends, we can simply use *MVar*s (Jones *et al.*, 1996) (Fig. 20), because we have shared memory in a single node and don't require Eden's sophisticated communication channels. **MB: explain MVars**

```
{-# NOINLINE putUnsafe #-}
putUnsafe :: a -> MVar a
putUnsafe a = unsafePerformIO $ do
  mVar <- newEmptyMVar
  putMVar mVar a
  return mVar
instance (NFData a) => Future MVar a where
  put = arr putUnsafe
  get = arr takeMVar >>> arr unsafePerformIO
```

Figure 20: MVar instance of the Future typeclass for the Par Monad and Multicore Haskell backends.

Furthermore, in order for these *Future* types to fit with the *ArrowParallel* instances we gave earlier, we have to give the necessary *NFData* and *Trans* instances, the latter are only needed in Eden. Because *MVar* already ships with a *NFData* instance, we only have to supply two simple instances for our *RemoteData* type:

```
instance NFData (RemoteData a) where
  rnf = rnf . rd
instance Trans (RemoteData a)
```

The *Trans* instance does not have any functions declared as the default implementation suffices here.

Going back to our communication example we can use this *Future* concept in order to enable direct communications between the nodes in the following way: In a distributed environment, this gives us a communication scheme with messages going through the master

```
someCombinator :: (Arrow arr) ⇒ [arr a b] → [arr b c] → arr [a] [c]
someCombinator fs1 fs2 =
  parEvalN () (map (>>>put) fs1) >>>
  rightRotate >>>
  parEvalN () (map (get>>>) fs2)
```

Figure 21: The combinator from Fig. 17 in parallel.

node only if it is needed - similar to what is shown in the trace in Fig. 22.**OL: Fig. 3 is not really clear. Do Figs 2-3 with a lot of load?**

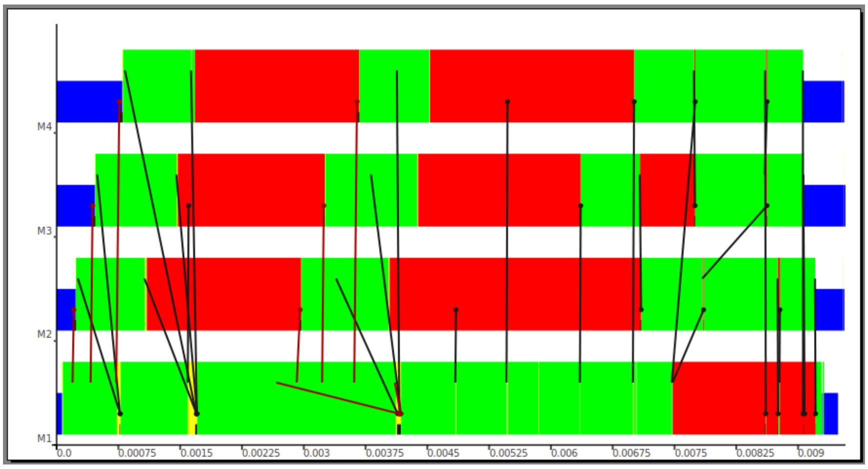


Figure 22: Communication between 4 threads with Futures. Other than in Fig. 18, threads communicate directly (black lines between the bars) instead of always going through the master node (bottom bar).

6 Map-based Skeletons

Now we have developed Parallel Arrows far enough to define some algorithmic skeletons useful to an application programmer.

6.1 Parallel map

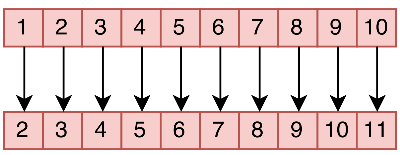


Figure 23: Schematic depiction of *parMap*.

The *parMap* skeleton (Fig. 23, 24) is probably the most common skeleton for parallel programs. We can implement it with *ArrowParallel* by repeating an arrow *arr a b* and then passing it into *parEvalN* to get an arrow *arr* $[a]$ $[b]$. Just like *parEvalN*, *parMap* is 100% strict.

$$\begin{aligned}&parMap :: (ArrowParallel\ arr\ a\ b\ conf) \Rightarrow conf \rightarrow (arr\ a\ b) \rightarrow (arr\ [a]\ [b])\\&parMap\ conf\ f = parEvalN\ conf\ (repeat\ f)\end{aligned}$$

Figure 24: Definition of parMap.

6.2 Lazy parallel map

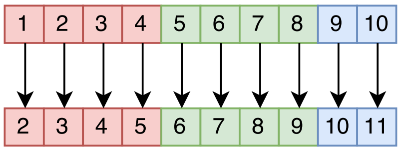


Figure 25: Schematic depiction of parMapStream.

As *parMap* (Fig. 23, 24) is 100% strict it has the same restrictions as *parEvalN* compared to *parEvalNLazy*. So it makes sense to also have a *parMapStream* (Fig. 25, 26) which behaves like *parMap*, but uses *parEvalNLazy* instead of *parEvalN*.

$$\begin{aligned}&parMapStream :: (ArrowParallel\ arr\ a\ b\ conf, ArrowChoice\ arr, ArrowApply\ arr) \Rightarrow\\&\quad conf \rightarrow ChunkSize \rightarrow arr\ a\ b \rightarrow arr\ [a]\ [b]\\&parMapStream\ conf\ chunkSize\ f = parEvalNLazy\ conf\ chunkSize\ (repeat\ f)\end{aligned}$$

Figure 26: Definition of *parMapStream*.

6.3 Statically load-balancing parallel map

A *parMap* (Fig. 23, 24) spawns every single computation in a new thread (at least for the instances of *ArrowParallel* we gave in this paper). This can be quite wasteful and a *farm* (Fig. 27, 28) that equally distributes the workload over *numCores* workers (if numCores is greater than the actual processor count, the fastest processor(s) to finish will get more tasks) seems useful. The definitions of helper functions *unshuffle*, *takeEach*, *shuffle* (shown in Appendix) originate from an Eden skeleton[6].

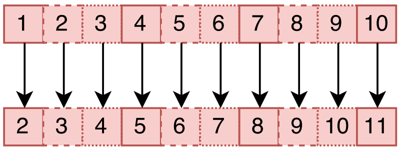


Figure 27: Schematic depiction of a *farm*, a statically load-balanced *parMap*.

```
farm :: (ArrowParallel arr a b conf,
    ArrowParallel arr [a] [b] conf, ArrowChoice arr) ⇒
    conf → NumCores → arr a b → arr [a] [b]
farm conf numCores f =
    unshuffle numCores >>>
    parEvalN conf (repeat (mapArr f)) >>>
    shuffle

unshuffle :: (Arrow arr) ⇒ Int → arr [a] [[a]]
unshuffle n = arr (λxs → [takeEach n (drop i xs) @|@ i ← [0..n−1]])

takeEach :: Int → [a] → [a]
takeEach n [] = []
takeEach n (x : xs) = x : takeEach n (drop (n−1) xs)

shuffle :: (Arrow arr) ⇒ arr [[a]] [a]
shuffle = arr (concat ∘ transpose)
```

Figure 28: The definition of *farm*.

6.4 The *farmChunk* Skeleton

Since a *farm* (Fig. 27, 28) is basically just *parMap* with a different work distribution, it is, again, 100% strict. So we can define *farmChunk* (Fig. 29, B 2) which uses *parEvalNLazy* instead of *parEvalN*. It is basically the same definition as for *farm*, with *parEvalN* replaced with *parEvalNLazy*, as Appendix shows.

6.5 Map and reduce

A simple *map–reduce* can be written like in Figure 30. Notice that the performance of the >>> combinator is essential for the performance of the skeleton. A definitive version would use Futures.

OL: it appears STRANGE. are the data really left alone and noded after map and taken from there by reduce? It makes sense only is no communication through master takes place. ELSE: CUT!

MB: this requires some work. Either change this to use futures or cut, yes.

OL: now rewritten as motivation for futures. maybe still cut?

[6] Available on Hackage under `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/src/Control-Parallel-Eden-Map.html`.

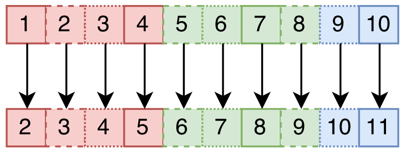


Figure 29: Schematic depiction of farmChunk.

$$
\begin{aligned}
&\mathit{parMapReduceDirect} :: (\mathit{ArrowParallel\ arr}\ [a]\ b\ \mathit{conf}, \mathit{ArrowApply\ arr}, \mathit{ArrowChoice\ arr}) \Rightarrow \\
&\quad \mathit{conf} \to \mathit{ChunkSize} \to \mathit{arr}\ a\ b \to \mathit{arr}\ (b,b)\ b \to b \to \mathit{arr}\ [a]\ b \\
&\mathit{parMapReduceDirect\ conf\ chunkSize\ mapfn\ foldfn\ neutral} = \\
&\quad \mathit{arr}\ (\mathit{chunksOf\ chunkSize}) >>> \\
&\quad \mathit{parMap\ conf}\ (\mathit{mapArr\ mapfn} >>> \mathit{foldlArr\ foldfn\ neutral}) >>> \\
&\quad \mathit{foldlArr\ foldfn\ neutral}
\end{aligned}
$$

Figure 30: Definition of parMapReduceDirect.

7 Topological Skeletons

Even though many algorithms can be expressed by parallel maps, some problems require more sophisticated skeletons. The Eden library leverages this problem and already comes[7] with more predefined skeletons, among them a *pipe*, a *ring*, and a *torus* implementations (Loogen, 2012). These seem like reasonable candidates to be ported to our Arrow-based parallel Haskell. We aim to showcase that we can express more sophisticated skeletons with Parallel Arrows as well.

7.1 Parallel pipe

The parallel *pipe* skeleton is semantically equivalent to folding over a list $[\mathit{arr}\ a\ a]$ of arrows with $>>>$, but does this in parallel, meaning that the arrows do not have to reside on the same thread/machine. We implement this skeleton using the *ArrowLoop* typeclass which gives us the $\mathit{loop} :: \mathit{arr}\ (a,b)\ (c,b) \to \mathit{arr}\ a\ c$ combinator which allows us to express recursive fix-point computations in which output values are fed back as input. For example **MB: das kann man hier so lassen, oder?OL: sicherlich!**

$$\mathit{loop}\ (\mathit{arr}\ (\lambda(a,b) \to (b, a : b)))$$

which is the same as

$$\mathit{loop}\ (\mathit{arr\ snd}\ \&\&\&\ \mathit{arr}\ (\mathit{uncurry}\ (:)))$$

defines an arrow that takes its input a and converts it into an infinite stream $[a]$ of it. Using this to our advantage gives us a first draft of a pipe implementation (Fig. 31) by plugging in the parallel evaluation call *parEvalN conf fs* inside the second argument of &&& and then only picking the first element of the resulting list with *arr last*.

[7] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html`.

$$
\begin{aligned}
&\mathit{pipeSimple} :: (\mathit{ArrowLoop\ arr}, \mathit{ArrowParallel\ arr\ a\ a\ conf}) \Rightarrow \\
&\quad \mathit{conf} \rightarrow [\mathit{arr\ a\ a}] \rightarrow \mathit{arr\ a\ a} \\
&\mathit{pipeSimple\ conf\ fs} = \\
&\quad \mathit{loop}\ (\mathit{arr\ snd}\ \&\&\& \\
&\qquad (\mathit{arr}\ (\mathit{uncurry}\ (:) >>> \mathit{lazy}) >>> \mathit{parEvalN\ conf\ fs})) >>> \\
&\quad \mathit{arr\ last} \\
&\mathit{lazy} :: (\mathit{Arrow\ arr}) \Rightarrow \mathit{arr}\ [a]\ [a] \\
&\mathit{lazy} = \mathit{arr}\ (\lambda{\sim}(x : \mathit{xs}) \rightarrow x : \mathit{lazy\ xs})
\end{aligned}
$$

Figure 31: A first implementation of the *pipe* skeleton expressed with Parallel Arrows. Note that the use of *lazy* is essential as without it programs using this definition would never halt. We need to enforce that the evaluation of the input $[a]$ terminates before passing it into *parEvalN*.

However, using this definition directly will make the master node a potential bottleneck in distributed environments as described in Section 5. Therefore, we introduce a more sophisticated version that internally uses Futures and get the final definition of *pipe*:

$$
\begin{aligned}
&\mathit{pipe} :: (\mathit{ArrowLoop\ arr}, \mathit{ArrowParallel\ arr}\ (\mathit{fut\ a})\ (\mathit{fut\ a})\ \mathit{conf}, \mathit{Future\ fut\ a}) \Rightarrow \\
&\quad \mathit{conf} \rightarrow [\mathit{arr\ a\ a}] \rightarrow \mathit{arr\ a\ a} \\
&\mathit{pipe\ conf\ fs} = \mathit{unliftFut}\ (\mathit{pipeSimple\ conf}\ (\mathit{map\ liftFut\ fs}))
\end{aligned}
$$

Sometimes, this *pipe* definition can be a bit inconvenient, especially if we want to pipe arrows of mixed types together, i.e. *arr a b* and *arr b c*. By wrapping these two arrows inside a common type we obtain *pipe2* (Fig. 32).

OL: I swapped the type classes here:

$$
\begin{aligned}
&\mathit{pipe2} :: (\mathit{ArrowLoop\ arr}, \mathit{ArrowChoice\ arr}, \mathit{Future\ fut}\ (([a],[b]),[c]), \\
&\quad \mathit{ArrowParallel\ arr}\ (\mathit{fut}\ (([a],[b]),[c]))\ (\mathit{fut}\ (([a],[b]),[c]))\ \mathit{conf}) \Rightarrow \\
&\quad \mathit{conf} \rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr\ b\ c} \rightarrow \mathit{arr\ a\ c} \\
&\mathit{pipe2\ conf\ f\ g} = \\
&\quad (\mathit{arr\ return}\ \&\&\&\ \mathit{arr}\ (\mathit{const}\ [\,]))\ \&\&\&\ \mathit{arr}\ (\mathit{const}\ [\,]) >>> \\
&\quad \mathit{pipe\ conf}\ (\mathit{replicate}\ 2\ (\mathit{unify\ f\ g})) >>> \\
&\quad \mathit{arr\ snd} >>> \mathit{arr\ head}\ \mathbf{where} \\
&\qquad \mathit{unify} :: (\mathit{ArrowChoice\ arr}) \Rightarrow \mathit{arr\ a\ b} \rightarrow \mathit{arr\ b\ c} \rightarrow \mathit{arr}\ (([a],[b]),[c])\ (([a],[b]),[c]) \\
&\qquad \mathit{unify\ f\ g} = \\
&\qquad\quad (\mathit{mapArr\ f} *** \mathit{mapArr\ g}) *** \mathit{arr}\ (\backslash\_ \rightarrow [\,]) >>> \\
&\qquad\quad \mathit{arr}\ (\lambda((a,b),c) \rightarrow ((c,a),b))
\end{aligned}
$$

Figure 32: Definition of *pipe2*.

Note that extensive use of this combinator over *pipe* with a hand-written combination data type will probably result in worse performance because of more communication overhead from the many calls to parEvalN. Nonetheless, we can define a parallel piping operator *parcomp* (Fig. 33, which is semantically equivalent to $>>>$ similarly to other parallel syntactic sugar from Section C.

OL: swapped type classes

$$(|>>>|) :: (ArrowLoop\ arr, ArrowChoice\ arr, Future\ fut\ (([a],[b]),[c]),$$
$$\quad ArrowParallel\ arr\ (fut\ (([a],[b]),[c]))\ (fut\ (([a],[b]),[c]))\ ()) \Rightarrow$$
$$\quad arr\ a\ b \rightarrow arr\ b\ c \rightarrow arr\ a\ c$$
$$(|>>>|) = pipe2\ ()$$

Figure 33: Definition of *parcomp*.

7.2 Ring skeleton

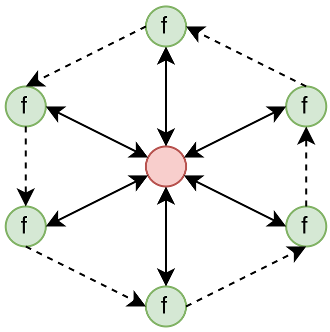


Figure 34: Schematic depiction of the ring skeleton.

Eden comes with a ring skeleton[8] (Fig. 34) implementation that allows the computation of a function $[i] \rightarrow [o]$ with a ring of nodes that communicate in a ring topology with each other. Its input is a node function $i \rightarrow r \rightarrow (o,r)$ in which r serves as the intermediary output that gets send to the neighbour of each node. This data is sent over direct communication channels, so called ‘remote data’. We depict it in Appendix, Fig. B 3.

We can rewrite this functionality easily with the use of *loop* as the definition of the node function, $arr\ (i,r)\ (o,r)$, after being transformed into an arrow, already fits quite neatly into the *loop*’s $arr\ (a,b)\ (c,b) \rightarrow arr\ a\ c$. In each iteration we start by rotating the intermediary input from the nodes $[fut\ r]$ with $second\ (rightRotate >>> lazy)$. Similarly to the *pipe* from Chapter 7.1 (Fig. 31), we have to feed the intermediary input into our *lazy* arrow here, or the evaluation would hang.**OL: meh, wording** The reasoning is explained by Loogen (2012):

> *Note that the list of ring inputs ringIns is the same as the list of ring outputs ringOuts rotated by one element to the right using the auxiliary function rightRotate. Thus, the program would get stuck without the lazy pattern, because the ring input will only be produced after process creation and process creation will not occur without the first input.*

Next, we zip the resulting $([i],[fut\ r])$ to $[(i, fut\ r)]$ with $arr\ (uncurry\ zip)$ so we can feed that into a our input arrow $arr\ (i,r)\ (o,r)$, which we transform into $arr\ (i, fut\ r)\ (o, fut\ r)$ before lifting it to $arr\ [(i, fut\ r)]\ [(o, fut\ r)]$ to get a list $[(o, fut\ r)]$. Finally we unzip this list into $([o],[fut\ r])$. Plugging this arrow $arr\ ([i],[fut\ r])\ ([o], fut\ r)$ into the definition of *loop*

[8] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html`

from earlier gives us $arr\ [i]\ [o]$, our ring arrow (Fig. 35). This combinator can, for example, be used to calculate the shortest paths in a graph using Warshall's algorithm. **OL: let's do it?**

$$
\begin{array}{l}
ring :: (ArrowLoop\ arr, Future\ fut\ r, ArrowParallel\ arr\ (i, fut\ r)\ (o, fut\ r)\ conf) \Rightarrow \\
\quad conf \rightarrow arr\ (i,r)\ (o,r) \rightarrow arr\ [i]\ [o] \\
ring\ conf\ f = \\
\quad loop\ (second\ (rightRotate >>> lazy) >>> arr\ (uncurry\ zip) >>> \\
\quad parMap\ conf\ (second\ get >>> f >>> second\ put) >>> arr\ unzip) \\
rightRotate :: (Arrow\ arr) \Rightarrow arr\ [a]\ [a] \\
rightRotate = arr\ \$\ \lambda list \rightarrow \textbf{case}\ list\ \textbf{of} \\
\quad [\,] \rightarrow [\,] \\
\quad xs \rightarrow last\ xs : init\ xs \\
lazy :: (Arrow\ arr) \Rightarrow arr\ [a]\ [a] \\
lazy = arr\ (\lambda{\sim}(x:xs) \rightarrow x : lazy\ xs)
\end{array}
$$

Figure 35: Final definition of the *ring* skeleton.

7.3 Torus skeleton

If we take the concept of a ring from 7.2 one dimension further, we get a torus (Fig. 36, 37). Every node sends ands receives data from horizontal and vertical neighbours in each communication round. With our Parallel Arrows we re-implement the *torus* combinator[9] from Eden—yet again with the help of the *ArrowLoop* typeclass.

Similar to the *ring*, we once again start by rotating the input, but this time not only in one direction, but in two. This means that the intermediary input from the neighbour nodes has to be stored in a tuple $([[fut\ a]], [[fut\ b]])$ in the second argument (loop only allows for two arguments) of our looped arrow $arr\ ([[c]], ([[fut\ a]], [[fut\ b]]))\ ([[d]], ([[fut\ a]], [[fut\ b]]))$ and our rotation arrow becomes $second\ ((mapArr\ rightRotate >>> lazy) *** (arr\ rightRotate >>>$

[9] Available on Hackage: `https://hackage.haskell.org/package/edenskel-2.0.0.2/docs/Control-Parallel-Eden-Topology.html`.

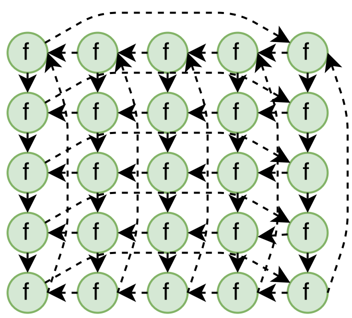


Figure 36: Schematic depiction of the *torus* skeleton.

lazy)) instead of the singular rotation in the ring as we rotate $[[\mathit{fut}\ a]]$ horizontally and $[[\mathit{fut}\ b]]$ vertically. Then, we once again zip the inputs for the input arrow with *arr* (*uncurry3 zipWith3 lazyzip3*) from $([[c]],([[\mathit{fut}\ a]],[[\mathit{fut}\ b]]))$ to $[[(c,\mathit{fut}\ a,\mathit{fut}\ b)]]$, which we then feed into our parallel execution.

This, however, is more complicated than in the ring case as we have one more dimension of inputs to be transformed. We first have to *shuffle* all the inputs to then pass it into *parMap conf* (*ptorus f*) which yields us $[(d,\mathit{fut}\ a,\mathit{fut}\ b)]$. We can then unpack this shuffled list back to its original ordering by feeding it into the specific unshuffle arrow we created one step earlier with *arr length* >>> *arr unshuffle* with the use of *app* :: *arr* (*arr a b*, *a*) *c* from the *ArrowApply* typeclass. Finally, we unpack this matrix $[[[(d,\mathit{fut}\ a,\mathit{fut}\ b)]]$ with *arr* (*map unzip3*) >>> *arr unzip3* >>> *threetotwo* to get $([[d]],([[\mathit{fut}\ a]],[[\mathit{fut}\ b]]))$.

OL: swapped type classes

```
torus :: (ArrowLoop arr, ArrowChoice arr, ArrowApply arr, Future fut a, Future fut b,
    ArrowParallel arr (c, fut a, fut b) (d, fut a, fut b) conf) ⇒
    conf → arr (c, a, b) (d, a, b) → arr [[c]] [[d]]
torus conf f =
    loop (second ((mapArr rightRotate >>> lazy) *** (arr rightRotate >>> lazy)) >>>
    arr (uncurry3 (zipWith3 lazyzip3)) >>>
    (arr length >>> arr unshuffle) &&& (shuffle >>> parMap conf (ptorus f)) >>> app >>>
    arr (map unzip3) >>> arr unzip3 >>> threetotwo)
ptorus :: (Arrow arr, Future fut a, Future fut b) ⇒
    arr (c, a, b) (d, a, b) → arr (c, fut a, fut b) (d, fut a, fut b)
ptorus f = arr (λ~(c, a, b) → (c, get a, get b)) >>> f >>> arr (λ~(d, a, b) → (d, put a, put b))
uncurry3 :: (a → b → c → d) → (a, (b, c)) → d
uncurry3 f (a, (b, c)) = f a b c
lazyzip3 :: [a] → [b] → [c] → [(a, b, c)]
lazyzip3 as bs cs = zip3 as (lazy bs) (lazy cs)
threetotwo :: (Arrow arr) ⇒ arr (a, b, c) (a, (b, c))
threetotwo = arr $ λ~(a, b, c) → (a, (b, c))
```

Figure 37: Definition of the *torus* skeleton.

As an example of using this skeleton (Loogen, 2012) showed the matrix multiplication using the Gentleman algorithm (Gentleman, 1978). Their instantiation of the skeleton *nodefunction* can be adapted as shown in Fig. 38. If we compare the trace from a call using

```
nodefunction :: Int → ((Matrix, Matrix), [Matrix], [Matrix]) → ([Matrix], [Matrix], [Matrix])
nodefunction n ((bA, bB), rows, cols) = ([bSum], bA : nextAs, bB : nextBs)
  where bSum = foldl′ matAdd (matMult bA bB) (zipWith matMult nextAs nextBs)
        nextAs = take (n − 1) rows
        nextBs = take (n − 1) cols
```

Figure 38: Adapted *nodefunction* for matrix multiplication with the *torus* from Fig. 37.

our arrow definition of the torus (Fig. 39) with the Eden version (Fig. 40) we can see that the behaviour of the arrow version is comparable.**OL: much more details on this!**

OL: more nodes!!!

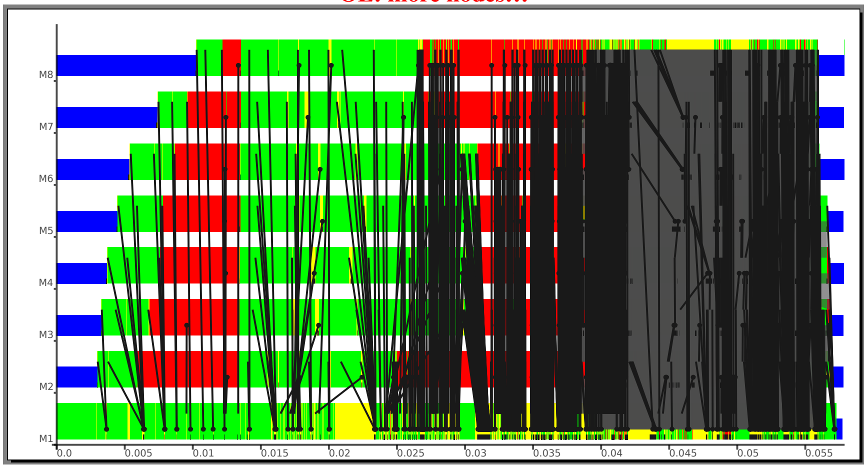


Figure 39: Matrix Multiplication with a torus (PArrows).

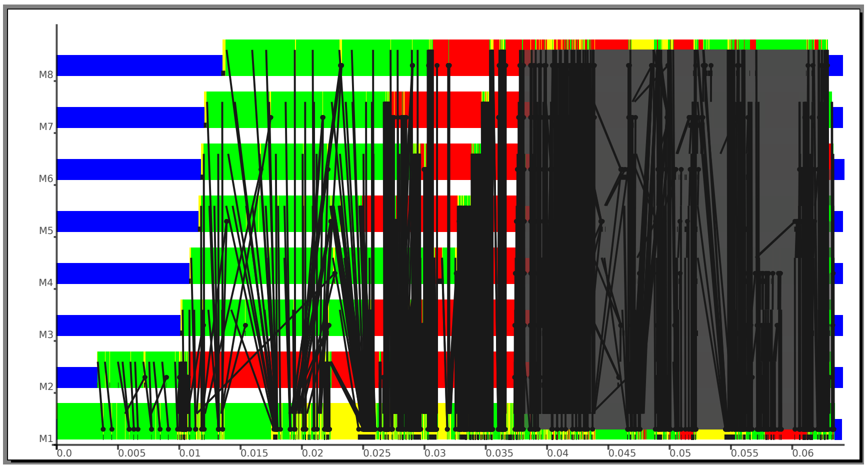
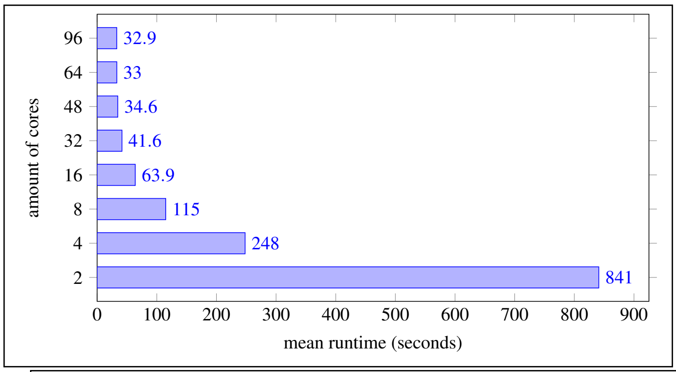
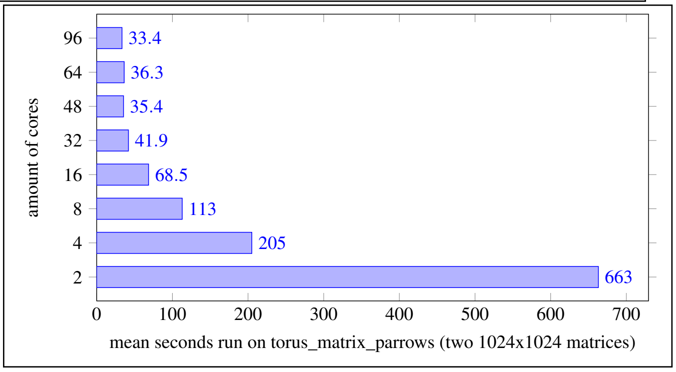


Figure 40: Matrix Multiplication with a torus (Eden).

8 Benchmarks





9 Conclusion

Arrows are a generic concept that allows for powerful composition combinators. To our knowledge we are the first ones to represent parallel computation with arrows.**OL: that strange arrows-based robot interaction paper from 1993 or so! clearly discuss in related work done!**

Arrows turn out to be a useful tool for composing in parallel programs. We do not have to introduce new monadic types that wrap the computation. Instead use arrows just like regular sequential pure functions. This work features multiple parallel backends: the already available parallel Haskell flavours. Parallel Arrows feature an implementation of the *ArrowParallel* interface for Multicore Haskell, *Par* Monad, and Eden. With our approach parallel programs can be ported across these flavours with no effort. Performancewise, Parallel Arrows are on par with existing parallel Haskells, as they do not introduce any notable overhead.**OL: PROOFS. Many proofs in benchmarks!**

MB: ArrowApply (or equivalent) are needed because we basically want to be able to produce intermediary results, this is by definition of the parallel evaluation combinators

OL: Remove websites from citations, put them into footnotes!

OL: Parrows + accelerate = love? Metion port to Frege. Mention the Par monad troubles.

9.1 Future Work

Our PArrows interface can be expanded to futher parallel Haskells. More specifically we target HdpH (Maier *et al.*, 2014) a modern distributed Haskell that would benefit from our Arrows notation. Future-aware special versions of Arrow combinated can be extended and further improved. We would look into more transparency of the API, it should basically infuse as little overhead as possible.

Of course, definitions of further skeletons are viable and needed. We are looking into more experiences with seamless porting of parallel PArrow-based programs across the backends.

Accelerate (Chakravarty *et al.*, 2011) is not related to our approach. It would be interesting to see a hybrid of both APIs.

OL: replace API with DSL globally?

References

Achten, Peter, van Eekelen, Marko, de Mol, Maarten, & Plasmeijer, Rinus. (2007). An arrow based semantics for interactive applications. *Draft Proceedings of the Symposium on Trends in Functional Programming*. TFP '07.

Achten, PM, van Eekelen, Marko CJD, Plasmeijer, MJ, & Weelden, A van. (2004). *Arrows for generic graphical editor components*.

Alimarine, Artem, Smetsers, Sjaak, van Weelden, Arjen, van Eekelen, Marko, & Plasmeijer, Rinus. (2005). There and back again: Arrows for invertible programming. *Pages 86–97 of: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell '05. ACM.

Aljabri, Malak, Loidl, Hans-Wolfgang, & Trinder, Phil W. (2014). The design and implementation of GUMSMP: A multilevel parallel haskell implementation. *Pages 37:37–37:48 of: Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*. IFL '13. ACM.

Aljabri, Malak, Loidl, Hans-Wolfgang, & Trinder, Phil. (2015). *Balancing shared and distributed heaps on NUMA architectures*. Springer. Pages 1–17.

Alt, Martin, & Gorlatch, Sergei. (2003). Future-Based RMI: Optimizing compositions of remote method calls on the Grid. *Pages 682–693 of:* Kosch, Harald, Böszörményi, László, & Hellwagner, Hermann (eds), *Euro-Par 2003*. LNCS 2790. Springer-Verlag.

Aswad, Mustafa, Trinder, Phil, Al Zain, Abdallah, Michaelson, Greg, & Berthold, Jost. (2009). Low pain vs no pain multi-core Haskells. *Pages 49–64 of: Trends in Functional Programming*.

Atkey, Robert. (2011). What is a categorical model of arrows? *Electronic notes in theoretical computer science*, **229**(5), 19–37.

Berthold, Jost. (2008). *Explicit and implicit parallel functional programming — concepts and implementation*. Ph.D. thesis, Philipps-Universität Marburg.

Berthold, Jost, & Loogen, Rita. (2006). Skeletons for recursively unfolding process topologies. Joubert, Gerhard R., Nagel, Wolfgang E., Peters, Frans J., Plata, Oscar G., Tirado, P., & Zapata, Emilio L. (eds), *Parallel Computing: Current & Future Issues of High-End Computing, ParCo 2005, Malaga, Spain*. NIC Series 33. Central Institute for Applied Mathematics, Jülich, Germany.

Berthold, Jost, & Loogen, Rita. (2007). Visualizing Parallel Functional Program Executions: Case Studies with the Eden Trace Viewer. *ParCo '07. Parallel Computing: Architectures, Algorithms and Applications*. IOS Press.

Berthold, Jost, Dieterle, Mischa, Lobachev, Oleg, & Loogen, Rita. (2009a). Distributed Memory Programming on Many-Cores – A Case Study Using Eden Divide-&-Conquer Skeletons. *Pages 47–55 of:* Großpitsch, K.-E., Henkersdorf, A., Uhrig, S., Ungerer, T., & Hähner, J. (eds), *Workshop on Many-Cores at ARCS '09 – 22nd International Conference on Architecture of Computing Systems 2009*. VDE-Verlag.

Berthold, Jost, Dieterle, Mischa, & Loogen, Rita. (2009b). Implementing parallel Google map-reduce in Eden. *Pages 990–1002 of:* Sips, Henk, Epema, Dick, & Lin, Hai-Xiang (eds), *Euro-Par 2009 Parallel Processing*. LNCS 5704. Springer Berlin Heidelberg.

Berthold, Jost, Dieterle, Mischa, Lobachev, Oleg, & Loogen, Rita. (2009c). *Parallel FFT with Eden skeletons*. PaCT '09. Springer. Pages 73–83.

Berthold, Jost, Loidl, Hans-Wolfgang, & Hammond, Kevin. (2016). PAEAN: Portable and scalable runtime support for parallel Haskell dialects. *Journal of functional programming*, **26**.

Botorog, G. H., & Kuchen, H. (1996). *Euro-Par'96 Parallel Processing*. LNCS 1123. Springer-Verlag. Chap. Efficient parallel programming with algorithmic skeletons, pages 718–731.

Chakravarty, Manuel M. T., Leshchinskiy, Roman, Peyton Jones, Simon L., Keller, Gabriele, & Marlow, Simon. (2007). Data Parallel Haskell: a status report. *Pages 10–18 of: DAMP '07*. ACM Press.

Chakravarty, Manuel M.T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., & Grover, Vinod. (2011). Accelerating Haskell array codes with multicore GPUs. *Pages 3–14 of: Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. ACM.

Clifton-Everest, Robert, McDonell, Trevor L, Chakravarty, Manuel M T, & Keller, Gabriele. (2014). Embedding Foreign Code. *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*. LNCS. Springer-Verlag.

Cole, M. I. (1989). Algorithmic skeletons: Structured management of parallel computation. *Research Monographs in Parallel and Distributed Computing*. Pitman.

Dagand, Pierre-Évariste, Kostić, Dejan, & Kuncak, Viktor. (2009). Opis: Reliable distributed systems in OCaml. *Pages 65–78 of: Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI '09. ACM.

Danelutto, M., Pasqualetti, F., & Pelagatti, S. (1997). Skeletons for Data Parallelism in P^3L. *Pages 619–628 of:* Lengauer, C., Griebl, M., & Gorlatch, S. (eds), *Euro-Par'97*. LNCS 1300. Springer-Verlag.

Darlington, J., Field, AJ, Harrison, PG, Kelly, PHJ, Sharp, DWN, Wu, Q., & While, RL. (1993). Parallel programming using skeleton functions. *Pages 146–160 of: Parallel architectures and languages Europe*. Springer-Verlag.

de la Encina, Alberto, Hidalgo-Herrero, Mercedes, Rabanal, Pablo, & Rubio, Fernando. (2011). *A parallel skeleton for genetic algorithms*. IWANN '11. Springer. Pages 388–395.

Dieterle, M., Horstmeyer, T., & Loogen, R. (2010a). Skeleton composition using remote data. *Pages 73–87 of:* Carro, M., & Peña, R. (eds), *12th International Symposium on Practical Aspects of Declarative Languages*. PADL '10, vol. 5937. Springer-Verlag.

Dieterle, M., Horstmeyer, T., Loogen, R., & Berthold, J. (2016). Skeleton composition versus stable process systems in Eden. *Journal of functional programming*, **26**.

Dieterle, Mischa, Berthold, Jost, & Loogen, Rita. (2010b). *A skeleton for distributed work pools in Eden*. FLOPS '10. Springer. Pages 337–353.

Dieterle, Mischa, Horstmeyer, Thomas, Berthold, Jost, & Loogen, Rita. (2013). *Iterating skeletons*. IFL '12. Springer. Pages 18–36.

Foltzer, Adam, Kulkarni, Abhishek, Swords, Rebecca, Sasidharan, Sajith, Jiang, Eric, & Newton, Ryan. (2012). A meta-scheduler for the Par-monad: Composable scheduling for the heterogeneous cloud. *Sigplan not.*, **47**(9), 235–246.

Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., & Mohr, B. (2010). The Scalasca performance toolset architecture. *Concurrency and computation: Practice and experience*, **22**(6).

Gentleman, W. Morven. (1978). Some complexity results for matrix computations on parallel processors. *Journal of the acm*, **25**(1), 112–115.

Gorlatch, Sergei. (1998). Programming with divide-and-conquer skeletons: A case study of FFT. *Journal of supercomputing*, **12**(1-2), 85–97.

Gorlatch, Sergei, & Bischof, Holger. (1998). A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel processing letters*, **8**(4).

Hammond, Kevin, Berthold, Jost, & Loogen, Rita. (2003). Automatic skeletons in Template Haskell. *Parallel processing letters*, **13**(03), 413–424.

Harris, Tim, Marlow, Simon, Peyton Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. ACM.

Horstmeyer, Thomas, & Loogen, Rita. (2013). Graph-based communication in Eden. *Higher-order and symbolic computation*, **26**(1), 3–28.

Huang, Liwen, Hudak, Paul, & Peterson, John. (2007). *HPorter: Using arrows to compose parallel processes*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 275–289.

Hudak, Paul, Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003). *Arrows, robots, and functional reactive programming*. Springer. Pages 159–187.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1–3), 67–111.

Hughes, John. (2005a). *Programming with arrows*. AFP '04. Springer. Pages 73–129.

Hughes, John. (2005b). *Programming with arrows*. Berlin, Heidelberg: Springer Berlin Heidelberg. Pages 73–129.

Jacobs, Bart, Heunen, Chris, & Hasuo, Ichiro. (2009). Categorical semantics for arrows. *Journal of functional programming*, **19**(3-4), 403–438.

Janjic, Vladimir, Brown, Christopher Mark, Neunhoeffer, Max, Hammond, Kevin, Linton, Stephen Alexander, & Loidl, Hans-Wolfgang. (2013). Space exploration using parallel orbits: a study in parallel symbolic computing. *Parallel computing*.

Jones, Simon Peyton, Gordon, Andrew, & Finne, Sigbjorn. (1996). Concurrent haskell. *Pages 295–308 of: POPL*, vol. 96.

Keller, Gabriele, Chakravarty, Manuel M.T., Leshchinskiy, Roman, Peyton Jones, Simon, & Lippmeier, Ben. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *Sigplan not.*, **45**(9), 261–272.

Kuper, Lindsey, Todd, Aaron, Tobin-Hochstadt, Sam, & Newton, Ryan R. (2014). Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. *Sigplan not.*, **49**(6), 2–14.

Lengauer, Christian, Gorlatch, Sergei, & Herrmann, Christoph. (1997). The static parallelization of loops and recursions. *The journal of supercomputing*, **11**(4), 333–353.

Li, Peng, & Zdancewic, S. (2006). Encoding information flow in Haskell. *Pages 12–16 of: 19th IEEE Computer Security Foundations Workshop*. CSFW '06.

Li, Peng, & Zdancewic, Steve. (2010). Arrows for secure information flow. *Theoretical computer science*, **411**(19), 1974–1994.

Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic notes in theoretical computer science*, **229**(5), 97–117.

Linton, S., Hammond, K., Konovalov, A., Al Zain, A. D., Trinder, P., Horn, P., & Roozemond, D. (2010). Easy composition of symbolic computation software: a new lingua franca for symbolic computation. *Pages 339–346 of: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. ISSAC '10. ACM Press.

Liu, Hai, Cheng, Eric, & Hudak, Paul. (2009). Causal commutative arrows and their optimization. *Sigplan not.*, **44**(9), 35–46.

Lobachev, Oleg. (2011). *Implementation and evaluation of algorithmic skeletons: Parallelisation of computer algebra algorithms*. Ph.D. thesis, Philipps-Universität Marburg.

Lobachev, Oleg. (2012). *Parallel computation skeletons with premature termination property*. FLOPS 2012. Springer. Pages 197–212.

Loogen, Rita. (2012). *Eden – parallel functional programming with Haskell*. Springer. Pages 142–206.

Loogen, Rita, Ortega-Mallén, Yolanda, & Peña-Marí, Ricardo. (2005). Parallel Functional Programming in Eden. *Journal of Functional Programming*, **15**(3), 431–475. Special Issue on Functional Approaches to High-Performance Parallel Programming.

Maier, Patrick, Stewart, Robert, & Trinder, Phil. (2014). The HdpH DSLs for scalable reliable computation. *Sigplan not.*, **49**(12), 65–76.

Mainland, Geoffrey, & Morrisett, Greg. (2010). Nikola: Embedding compiled GPU functions in Haskell. *Sigplan not.*, **45**(11), 67–78.

Marlow, S., Peyton Jones, S., & Singh, S. (2009). Runtime support for multicore Haskell. *Acm sigplan notices*, **44**(9), 65–78.

Marlow, Simon. (2013). *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. "O'Reilly Media, Inc.".

Marlow, Simon, Newton, Ryan, & Peyton Jones, Simon. (2011a). A monad for deterministic parallelism. *Sigplan not.*, **46**(12), 71–82.

Marlow, Simon, Newton, Ryan, & Peyton Jones, Simon. (2011b). A monad for deterministic parallelism. *Sigplan not.*, **46**(12), 71–82.

McDonell, Trevor L., Chakravarty, Manuel M. T., Grover, Vinod, & Newton, Ryan R. (2015). Type-safe runtime code generation: Accelerate to LLVM. *Sigplan not.*, **50**(12), 201–212.

Nikhil, R., & Arvind, L. A. (2001). *Implicit parallel programming in pH*. Morgan Kaufmann.

Paterson, Ross. (2001). A new notation for arrows. *Sigplan not.*, **36**(10), 229–240.

Perfumo, Cristian, Sönmez, Nehir, Stipic, Srdjan, Unsal, Osman, Cristal, Adrián, Harris, Tim, & Valero, Mateo. (2008). The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. *Pages 67–78 of: Proceedings of the 5th Conference on Computing Frontiers*. CF '08. ACM.

Rabhi, F. A., & Gorlatch, S. (eds). (2003). *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag.

Russo, Alejandro, Claessen, Koen, & Hughes, John. (2008). A library for light-weight information-flow security in Haskell. *Pages 13–24 of: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*. Haskell '08. ACM.

Stewart, Robert, Maier, Patrick, & Trinder, Phil. (2016). Transparent fault tolerance for scalable functional computation. *Journal of functional programming*, **26**.

Svensson, Joel. (2011). *Obsidian: Gpu kernel programming in haskell*. Ph.D. thesis, Chalmers University of Technology.

Trinder, Phil W., Hammond, Kevin, Mattson Jr., James S., Partridge, Andrew S., & Peyton Jones, Simon L. (1996). GUM: a Portable Parallel Implementation of Haskell. *PLDI'96*. ACM Press.

Trinder, P.W., Hammond, K., Loidl, H-W., & Peyton Jones, S. (1998). Algorithm + Strategy = Parallelism. *Journal of functional programming*, **8**(1), 23–60.

Vizzotto, Juliana, Altenkirch, Thorsten, & Sabry, Amr. (2006). Structuring quantum effects: superoperators as arrows. *Mathematical structures in computer science*, **16**(3), 453–468.

Wheeler, K. B., & Thain, D. (2009). Visualizing massively multithreaded applications with ThreadScope. *Concurrency and computation: Practice and experience*, **22**(1), 45–67.

A Utility Functions

To be able to go into detail on parallel arrows, we introduce some utility combinators first, that will help us later: *map*, *foldl* and *zipWith* on arrows.

The *mapArr* combinator (Fig. A 1) lifts any arrow *arr a b* to an arrow *arr* $[a]$ $[b]$ (Hughes, 2005b). Similarly, we can also define *foldlArr* (Fig. A 2) that lifts any arrow *arr* (b,a) *b* with a neutral element *b* to *arr* $[a]$ *b*.

```
mapArr :: ArrowChoice arr => arr a b -> arr [a] [b]
mapArr f =
  arr listcase >>>
  arr (const []) ||| (f *** mapArr f >>> arr (uncurry (:)))
listcase [] = Left ()
listcase (x:xs) = Right (x,xs)
```

Figure A 1: The definition of *map* over Arrows and the *listcase* helper function.

Finally, with the help of *mapArr* (Fig. A 1), we can define *zipWithArr* (Fig. A 3) that lifts any arrow *arr* (a,b) *c* to an arrow *arr* $([a],[b])$ $[c]$.

These combinators make use of the *ArrowChoice* type class which provides the $|||$ combinator. It takes two arrows *arr a c* and *arr b c* and combines them into a new arrow

$$
\begin{aligned}
&\mathit{foldlArr} :: (\mathit{ArrowChoice}\ \mathit{arr}, \mathit{ArrowApply}\ \mathit{arr}) \Rightarrow \mathit{arr}\ (b,a)\ b \to b \to \mathit{arr}\ [a]\ b\\
&\mathit{foldlArr}\ f\ b =\\
&\quad \mathit{arr}\ \mathit{listcase} >\!>\!>\\
&\quad \mathit{arr}\ (\mathit{const}\ b)\ |||\\
&\qquad (\mathit{first}\ (\mathit{arr}\ (\lambda a \to (b,a)) >\!>\!> f >\!>\!> \mathit{arr}\ (\mathit{foldlArr}\ f)) >\!>\!> \mathit{app})
\end{aligned}
$$

Figure A 2: The definition of *foldl* over Arrows.

$$
\begin{aligned}
&\mathit{zipWithArr} :: \mathit{ArrowChoice}\ \mathit{arr} \Rightarrow \mathit{arr}\ (a,b)\ c \to \mathit{arr}\ ([a],[b])\ [c]\\
&\mathit{zipWithArr}\ f = (\mathit{arr}\ \$\ \lambda(\mathit{as},\mathit{bs}) \to \mathit{zipWith}\ (,)\ \mathit{as}\ \mathit{bs}) >\!>\!> \mathit{mapArr}\ f
\end{aligned}
$$

Figure A 3: *zipWith* over arrows.

$\mathit{arr}\ (\mathit{Either}\ a\ b)\ c$ which pipes all *Left a*'s to the first arrow and all *Right b*'s to the second arrow.

$$(|||) :: \mathit{ArrowChoice}\ \mathit{arr}\ a\ c \to \mathit{arr}\ b\ c \to \mathit{arr}\ (\mathit{Either}\ a\ b)\ c$$

With the zipWithArr combinator we can also write a combinator *listApp*, that lifts a list of arrows $[\mathit{arr}\ a\ b]$ to an arrow $\mathit{arr}\ [a]\ [b]$.

$$
\begin{aligned}
&\mathit{listApp} :: (\mathit{ArrowChoice}\ \mathit{arr}, \mathit{ArrowApply}\ \mathit{arr}) \Rightarrow [\mathit{arr}\ a\ b] \to \mathit{arr}\ [a]\ [b]\\
&\mathit{listApp}\ \mathit{fs} = (\mathit{arr}\ \$\ \lambda \mathit{as} \to (\mathit{fs},\mathit{as})) >\!>\!> \mathit{zipWithArr}\ \mathit{app}
\end{aligned}
$$

Note that this additionally makes use of the *ArrowApply* typeclass that allows us to evaluate arrows with $\mathit{app} :: \mathit{arr}\ (\mathit{arr}\ a\ b, a)\ c$.

B Omitted Funtion Definitions

We have omitted some function definitions in the main text for brevity, and redeem this here. We warp Eden's build-in Futures in PArrows as in Figure B 1. The full definition of *farmChunk* is in Figure B 2. Eden definition of *ring* skeleton following (Loogen, 2012) is in Figure B 3.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{RemoteData}\ a = \mathit{RD}\ \{\mathit{rd} :: \mathit{RD}\ a\}\\
&\mathbf{instance}\ (\mathit{Trans}\ a) \Rightarrow \mathit{Future}\ \mathit{RemoteData}\ a\ \mathbf{where}\\
&\quad \mathit{put} = \mathit{arr}\ (\lambda a \to \mathit{RD}\ \{\mathit{rd} = \mathit{release}\ a\})\\
&\quad \mathit{get} = \mathit{arr}\ \mathit{rd} >\!>\!> \mathit{arr}\ \mathit{fetch}
\end{aligned}
$$

Figure B 1: *RD*-based *RemoteData* version of *Future* for the Eden backend.

Furthermore, *parEval2* (Fig. B 4) is achieved as follows: We start by transforming the (a,c) input into a two-element list $[\mathit{Either}\ a\ c]$ by first tagging the two inputs with *Left* and *Right* and wrapping the right element in a singleton list with *return* so that we can combine them with $\mathit{arr}\ (\mathit{uncurry}\ (:))$. Next, we feed this list into a parallel arrow running on two instances of $f +\!\!+\!\!+ g$ as described above. After the calculation is finished, we convert the

```
farmChunk :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
  ArrowChoice arr, ArrowApply arr) =>
  conf -> ChunkSize -> NumCores -> arr a b -> arr [a] [b]
farmChunk conf chunkSize numCores f =
  unshuffle numCores >>>
  parEvalNLazy conf chunkSize (repeat (mapArr f)) >>>
  shuffle
```

Figure B 2: Definition of *farmChunk*.

```
ringSimple :: (Trans i, Trans o, Trans r) => (i -> r -> (o, r)) -> [i] -> [o]
ringSimple f is = os
  where (os, ringOuts) = unzip (parMap (toRD $ uncurry f) (zip is $ lazy ringIns))
    ringIns = rightRotate ringOuts
toRD :: (Trans i, Trans o, Trans r) => ((i, r) -> (o, r)) -> ((i, RD r) -> (o, RD r))
toRD f (i, ringIn) = (o, release ringOut)
  where (o, ringOut) = f (i, fetch ringIn)
rightRotate :: [a] -> [a]
rightRotate [] = []
rightRotate xs = last xs : init xs
lazy :: [a] -> [a]
lazy ~(x : xs) = x : lazy xs
```

Figure B 3: Eden's definition of the *ring* skeleton.

resulting $[Either\ b\ d]$ into $([b],[d])$ with *arr partitionEithers*. The two lists in this tuple contain only one element each by construction, so we can finally just convert the tuple to (b,d) in the last step.

```
parEval2 :: (ArrowChoice arr,
  ArrowParallel arr (Either a c) (Either b d) conf) =>
  conf -> arr a b -> arr c d -> arr (a, c) (b, d)
parEval2 conf f g =
  arr Left *** (arr Right >>> arr return) >>>
  arr (uncurry (:)) >>>
  parEvalN conf (replicate 2 (f +++ g)) >>>
  arr partitionEithers >>>
  arr head *** arr head
```

Figure B 4: Definition of parEval2.

C Syntactic Sugar

For basic arrows, we have the $***$ combinator (Fig. 10) which allows us to combine two arrows *arr a b* and *arr c d* into an arrow *arr* (a,c) (b,d) which does both computations at once. This can easily be translated into a parallel version $***$ with the use of *parEval2*, but for this we require a backend which has an implementation that does not require any configuration (hence the $()$ as the *conf* parameter):

$$
\begin{aligned}
&(|***|) :: (\mathit{ArrowChoice\ arr}, \mathit{ArrowParallel\ arr}\ (\mathit{Either\ a\ c})\ (\mathit{Either\ b\ d})\ ())) \Rightarrow \\
&\quad \mathit{arr\ a\ b} \rightarrow \mathit{arr\ c\ d} \rightarrow \mathit{arr}\ (a,c)\ (b,d) \\
&(|***|) = \mathit{parEval2}\ ()
\end{aligned}
$$

We define the parallel &&& in a similar manner to its sequential pendant &&& (Fig. 10):

$$
\begin{aligned}
&(|\&\&\&|) :: (\mathit{ArrowChoice\ arr}, \mathit{ArrowParallel\ arr}\ (\mathit{Either\ a\ a})\ (\mathit{Either\ b\ c})\ ()) \Rightarrow \\
&\quad \mathit{arr\ a\ b} \rightarrow \mathit{arr\ a\ c} \rightarrow \mathit{arr\ a}\ (b,c) \\
&(|\&\&\&|)\ f\ g = (\mathit{arr}\ \$\ \lambda a \rightarrow (a,a)) >>> f\ |***|\ g
\end{aligned}
$$