Chapter 1

Graph-based Communication in Eden

Thomas Horstmeyer and Rita Loogen[1]
Category: Research

Abstract: We present a new approach to the definition and creation of process topologies in the parallel functional Haskell extension Eden. Grace (Graph-based communication in Eden) allows a programmer to specify a network of processes as a graph, where the graph nodes represent processes and the edges represent communication channels. Thus, the specification and creation of complex communication topologies is simplified a lot. The main benefit of the new approach is a clean separation between coordination and computation. Runtime experiments show that Grace has a marginal overhead in comparison with traditional Eden code.

1.1 INTRODUCTION

The parallel functional language Eden [7] enables programmers to define process networks with arbitrary topologies. However, the creation of a non-tree-like topology had up to now to be done on a low level of abstraction using so-called dynamic channels. These channels are created by receiver processes and must be passed to the corresponding sender processes to establish a direct channel connection between those processes. This is a rather tedious and error-prone task.

In this paper, we present a new approach to the definition and creation of process topologies in Eden. Grace (Graph-based communication in Eden) allows a programmer to specify a network of processes as a graph, where the graph nodes represent processes and the edges represent communication channels. The graph is described as a Haskell data structure `ProcessNetwork a`, where `a` is the

[1]Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Hans-Meerwein-Straße, 35032 Marburg, Germany;
`{horstmey,loogen}@informatik.uni-marburg.de`

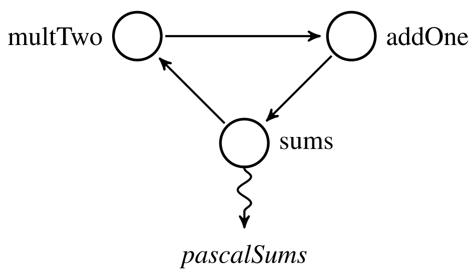

FIGURE 1.1. Network Topology for the Sums of Elements in the Pascal's Triangle

result type of the network computation. A function `start` will instantiate the network and automatically set up the corresponding process topology, i.e. the processes are created and the necessary communication channels are installed. The main benefit of the new approach is a clean separation between coordination and computation. The network specification encapsulates the coordinational aspects. The graph nodes are annotated with functions describing the computations of the corresponding processes.

Generally a user defines the process network by placing functions on nodes and connecting the nodes with edges. For every parameter that a function takes, the corresponding node must have an incoming edge. The third parameter of the edge constructor is used to define an ordering on these incoming edges to map them unambiguously to the parameters. The result computed on a node will be transmitted over every outgoing edge to other nodes. Since not every successor node might need the whole result an optional transformation function can be placed on edges that filters the data to be transmitted before the transfer. No filtering is expressed using `nothing`[2]. A small extension to the base system allows the definition of multiple incoming edges for a parameter that is a list which then will be received element-wise over these edges.

An Introductory Example

Let us take a look at a simple network that computes the sequence $\langle x_n | n \geq 1 \rangle$ with $x_1 = 1$ and $x_i = 2x_{i-1} + 1$ for all $i > 1$. Here, x_n gives you the sum of the elements in the Pascal's triangle with n levels. We use two separate processes to compute the multiplication and the addition. Figure 1.1 visualises the network.

Listing 1.1 shows how to describe this network with the help of Grace. It uses the data types and functions of the Grace package shown in Listing 1.2. The network is specified as a graph structure that is passed to the Grace function `build`. It consists of the node for the main process where the `sums` function is evaluated, two nodes labeled "mult" and "add" for the separate processes and edges connecting the nodes. The third and fourth parameter of the edge constructor `E` are not of interest in this small example. Applying the function `start` to the network

[2]`Nothing` of the `Maybe`-data type with a fixed type

Listing 1.1. Element Sums in the Pascal's Triangle with Grace

```
pascalSums = start network
  where
    addOne  = map (+1)      :: [Int] → [Int]
    multTwo = map (*2)      :: [Int] → [Int]
    sums    = λ xs → 1:xs :: [Int] → [Int]


    network :: ProcessNetwork [Int]
    network = build ("sum", sums) nodes edges

    nodes :: [Node String]
    nodes = [N "mult" multTwo, N "add" addOne]

    edges :: [Edge String Int]
    edges = [E "mult" "add"  0 nothing,
             E "add"  "sum"  0 nothing,
             E "sum"  "mult" 0 nothing]
```

Listing 1.2. Some Data Types and Functions of the Grace Package

```
data Node l = forall f a g r p. (Placeable f a g r p) ⇒ N l f

data Edge n l = forall a b p.
                (Trans a, Trans b, Placeable (a → b) a b b p) ⇒
                E n n l (Maybe (a → b))

build :: forall f a g r p n e. (Placeable f a g r p, Ord e, Eq n) ⇒
         (n, f) → [Node n] → [Edge n e] → ProcessNetwork r

start :: (Trans a) ⇒ ProcessNetwork a → a
```

will instantiate its processes, build the communication channels and compute the result.

Plan of Paper. The next section contains a short introduction into Eden. Basic constructs of Grace are explained in Section 1.3. Advanced constructs follow in Section 1.4. Implementation details are discussed in Section 1.5, while an experimental evaluation is presented in Section 1.6. Section 1.7 gives an implementation of the hyperquicksort algorithm that uses all of Grace's features. The paper finishes with a discussion of related work in Section 1.8 and conclusions in Section 1.9.

1.2 EDEN

The parallel Haskell dialect Eden [7] extends Haskell [8] with an explicit notion of processes (function applications evaluated remotely in parallel). The programmer has direct control over evaluation site, process granularity, data distribution and communication topology, but does not have to manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer.

The essential two coordination constructs of Eden are process abstraction and instantiation:

```
process :: (Trans a, Trans b) ⇒ (a → b)     → Process a b
( # )   :: (Trans a, Trans b) ⇒ Process a b → a → b
```

The function `process` embeds functions of type `a → b` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the `Trans` class of transmissible values. Evaluation of an expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`. The type class `Trans` provides overloaded communication functions for *lists*, which are transmitted as streams, element by element, and for *tuples*, which are evaluated component-wise by concurrent threads in the same process. An Eden process can thus contain a variable number of threads during its lifetime.

Two additional non-functional features of Eden are essential for performance optimisations and the creation of non-hierarchical process networks: nondeterministic stream merging and explicit communication. Eden's non-deterministic function `merge :: Trans a ⇒ [[a]] → [a]` merges a list of streams into a single stream and thus, provides many-to-one communication. Communication channels may be created implicitly during process creation – in this case we call them *static channels* – or explicitly during process evaluation. In the latter case we call them *dynamic channels*. The following functions provide the interface to create and use dynamic channels:

```
new     :: Trans a ⇒ (ChanName a → a → b) → b
parfill :: Trans a ⇒  ChanName a → a → b   → b
```

Evaluating `new (λ name val → e)`, a process creates a dynamic channel `name` of type `ChanName a` in order to receive a value `val` of type `a`. After creation, the channel should be passed to another process (just like normal data) inside the expression result `e`, which will as well use the eventually received value `val`. Evaluating `(parfill name e1 e2)` in the other process has the side-effect that a new thread is forked to concurrently evaluate and send the value `e1` via the channel. The overall result of the expression is `e2`.

Listing 1.3 shows a definition of the introductory example in Eden. The main process `pascalSums` creates process `addOne` which in turn creates process `multTwo`. Process `multTwo` creates a dynamic channel `chan` which it returns to its creator `addOne`. The latter simply passes this channel to the main process which uses the channel to pass the result list `(1:result)` directly to the process `multTwo`. Thus, the channels from `multTwo` to `addOne` and from `addOne` to the main process are implicitly created static channels, while the channel connection between the main process the `multTwo` is dynamically created.

1.3 BASIC CONSTRUCTS

With Grace the desired network topology is specified through a directed graph structure where nodes represent the processes and edges the communication be-

Listing 1.3. Element Sums in the Pascal Triangle in Eden

```
pascalSums :: [Int]
pascalSums = parfill chan (1:result) (1:result)
  where
    (chan, result) = (process addOne) # ()
    addOne () = (λ(c, ins) → (c, map (+1) ins))
                ((process multTwo) # ())
    multTwo () = new (λchan ins  →
                (chan, (map (*2) ins)))
```

tween them. On every node a function must be placed which will receive each of its arguments through an incoming edge. The functions' result will be sent to every successor node. To unambiguously map incoming edges to function parameters the edges must have a weight, i.e. a label for whose type an ordering is defined.

The graph itself is specified as a list of nodes, a list of edges and a separate special node, whose result is considered to be the result of the whole network. In the following we will refer to this special node as *main node*. A node has a label and a function placed on it.

```
data Node l = forall f a g r p.
              (Placeable f a g r p) ⇒
              N l f
```

This construct uses multi-parameter classes with functional dependencies [11] and explicit quantification to introduce the type variables `a`, `g`, `r` and `p` which are dependent on f. The class `Placeable` in the type context is used by the implementation. However, the user needs not declare a suitable instance – an instance will be derived automatically for every possible function. The only constraint is that the function may not have allquantified type variables. Note that the type variable `f` that represents the placed function is existentially quantified [5] and does not appear in the type of the node. This allows us to declare a list of nodes as a standard Haskell list `[Node l]` even if the functions placed on the nodes are of different type.

Edges consist of two nodes (from and to), a label and an optional function. The use of the latter will be explained in Section 1.4.

```
data Edge n l = forall a b p.
                (Trans a, Trans b,
                 Placeable (a → b) a b b p) ⇒
                E n n l (Maybe (a → b))
```

When nodes and edges have been specified they can be passed to the function `build` that combines them into an abstraction of a process network.

```
build :: forall f a g r p n e.
         (Placeable f a g r p, Ord e, Eq n) ⇒
         (n, f) →           -- main node
         [Node n] →         -- other nodes
         [Edge n e] →       -- edges
         ProcessNetwork r
```

The type context `Ord e` and `Eq n` ensures that edges can be ordered by their label and nodes can be identified by their label. The main node is not of type

`Node n` but a pair of its label and function because the existential quantification would hide `f` which is needed to determine the result type `r` of the network's computation . `Placeable f a g r p` relates `f` to `r`, such that `r` is a constant and `f` is a type $\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow r$ for $k \geq 0$.

The instantiation of the network and the computation of its result is executed by passing the process network to the function `start`.

```
start :: (Trans a) ⇒ ProcessNetwork a → a
```

The introductory example given in listing 1.1 shows the clear distinction between computation logic and topology specification. Due to the usage of strings as node labels the intended interrelations between the processes are even perceptible in the edge declarations.

1.4 ADVANCED CONSTRUCTS

Parameterized Number of Edges

With the basic constructs each node has exactly as many incoming edges as its function takes parameters. This is not very practical for processes that need to communicate with a high, parameterized number of other processes. Without Grace one would store the incoming data as elements in a list. An example is the master-worker skeleton [7, 6], where the master has a list of incoming streams that is merged with the Eden-function `merge`. To allow something similar in Grace we have made it possible not only to receive a list as a stream but also element-wise from different communication partners. This is realized by introducing a new data type `Lister f`, that can be placed on a node like an ordinary function. It is created using the function `lister`:

```
lister :: (IsFunctionType f flag,
          Placeable' f a g r p) ⇒
          f → [Int] → Lister f
```

Again, for the user the type context is not really important. Appropriate instances will be derived for any given function.

The list parameter specifies the behaviour for each of the functions arguments. If the i-th element of the list is 0 the corresponding parameter of the function will be treated normally. However, if it is $k > 0$ and the i-th parameter of the function is a list then exactly k channels will be created for this parameter when building the network. A single list element will be received over each of these channels.

See Section 1.6 for a Grace version of the master-worker skeleton.

Selection on Edges

In most of the cases where a node has multiple outgoing edges not all successors are really interested in the whole result that is computed on the node. It is quite common that the result, e.g. a tuple, is supposed to be distributed component-wise.

Eden's eager communication forces us to address this problem on the sender's side. We allow to place a function on an edge that is used to transform the node's result before it is communicated over that edge. The edge data type given in Section 1.3 already takes this possibility into account. This will typically be used for selection or filtering but technically arbitrary transformations are possible.

1.5 BEHIND THE SCENES – GRACE IMPLEMENTATION DETAILS

The implementation faces a few challenges, most of which can be solved using common language extensions like multi-parameter classes, functional dependencies and relaxations in type checking.

The easiest task is how to specify the graph. A list of nodes (and edges) that carry functions of different types must be made possible. Since the number of functions is not fixed we can not use an algebraic data type. The use of the HList-library [4] would be possible but we do not really need its advanced features. The user has to build the list, which should therefore be as easy as possible. By declaring the node data type as existential type that hides the type of the placed function ordinary Haskell lists can be used.

A more complicated problem is how to partition the user supplied function type into its parts, i.e. parameter types and result type, so that individual channels for these can be created. Here, we use techniques developed in the context of generic programming. We define a multi-parameter class with dependent types to make the parts of the function type accessible.

```
class (Trans argtype, Trans restype) ⇒
      Placeable ftype argtype remtype restype plisttype
       | ftype → argtype remtype restype plisttype
       where ...
```

The function's type `ftype` determines all the other types. The type of the function's first argument is `argtype`, `remtype` is the remaining part of the function's type without the first argument. The final result type of the function, which you get after applying all parameters is `restype`. Finally, `plisttype` is a type level list of all the parameters. This list uses the type constructors:

```
data PNilType = PNil
data PConsType a b = PCons a b
```

For a function of type `Int → Char → Bool` we would get the instance:

```
Placeable
  (Int → Char → Bool)                        -- ftype
  Int                                        -- argtype
  (Char → Bool)                              -- remtype
  Bool                                       -- restype
  (PCons Int (Pcons Char (PCons Bool PNil)))  -- plisttype
```

The type context `(Trans argtype, Trans restype)` ensures that both first argument and result can be transported over Eden-channels. For the other parameters we will ensure this via recursive instance declarations.

Let us now take a look at these instance declarations. We do not show the (not so interesting) class' methods but it is important for these (and for the recursive structure of the instance declarations as well) to be able to distinguish between constants and functions that take parameters. To decide this, we follow the technique described by Kiselyov on his website [3], based on the class `TypeCast` from the HList-library.

We declare a class `IsFunctionType a b` that relates a given type to one of the types `HFuncListParam`, `HFuncConstParam` and `HConst` if the type is a function type that takes a list as first parameter, a function type that takes no list as first parameter or is a constant, respectively. The distinction between list and non-list parameters is needed to support the `Lister`-construct.

Originally we had intended to only give one instance declaration for the class `Placeable`:

```
instance (IsFunctionType ftype, flag,
          Placeable' flag ftype argtype
                        remtype restype plisttype) ⇒
         Placeable ftype argtype remtype
                   restype plisttype
         where ...
```

Any method in this class would redirect its call to a corresponding method in the class `Placeable'`. For `Placeable'` we give instances for any of the three possible flags as shown in listing 1.4.

Listing 1.4. Instances of `Placeable`

```
instance (Trans argtype,
          Placeable' flag remtype a g restype plisttype,
          IsFunctionType remtype flag)
          ⇒ Placeable' HFuncConstParam
                      (argtype → remtype)
                      argtype remtype restype
                      (PConsType argtype plisttype)
    where ...

instance (Trans argelemtype,
          Placeable' flag remtype a g restype plisttype,
          IsFunctionType remtype flag)
          ⇒ Placeable' HFuncListParam
                      ([argelemtype] → remtype)
                      [argelemtype] remtype restype
                      (PConsType [argelemtype] plisttype)

    where ...

instance (Trans ftype) ⇒ Placeable' HConst
                                   ftype
                                   () () ftype
                                   PNilType
    where ...
```

You can see, that for the constant function the result type is the same as the function type.

In the end, the `Lister`-data type got its own, second instance declaration. This ensures that `Lister` can only be 'wrapped' around a function and not be another part of a function type, e.g. `Lister ([Int] → Int)` has an instance but `Int → Lister ([Int] → Int)` has not.

While the afore mentioned challenges all could be addressed, a serious hen-and-egg-problem is yet only partly solved. The dynamic channels specified by the edges must be created by the receiving process and communicated – via other channels – to the sender. We chose a star network to accomplish this: Every node sends its channels to the main node that distributes them where they belong. The problem is the typing: A channel's type is determined by the communicated data's type. A channel of our building network transports an arbitrary number of channels with user-defined types.

Since we do not want to pass this problem to the user we do not see any other way than cheating: hide the channels type for transport using `unsafeCoerce` and reestablish the type afterwards with the same operation. However, the reapplied type could be a different one if the process network was erroneous. The channel is created with the type of data that is expected on the receiver's side. The sender later casts it to a type for the data it intends to send. If these do not match the computation may lead to unexpected results and even deadlocks. Unfortunately, the type safety is broken at this point.

We hope to reintroduce type safety in the future. Theoretically it should be possible to use the specified graph for type checking. Template Haskell [9] might be a tool suited to address this task.

We have considered runtime type checking using dynamic typing as well. But on the one hand it creates a data overhead at runtime that we want to avoid, and on the other hand the existing solutions are not designed to work when data migrates over different runtime environments.

To support debugging we have written a function that verifies the topology of the network and checks if every node has as many incoming edges as its function's arity.

1.6 EXAMPLE AND RUNTIME BEHAVIOUR

Grace's strength lies in the easy creation of complex networks. This of course introduces a certain overhead. To examine this overhead we have chosen a classic application that builds a tree-shaped network and implemented a version with Grace. This was compared against an existing version that does not use Grace. The latter uses only implicit channels and has therefore the smallest possible communication overhead. If Grace performs well in this case then we can expect good results in cases where Grace has the advantage of a more shallow communication structure during network creation as well.

The core of the program is the classic master-worker-skeleton in a slight variation that returns unmerged data. The application computes a visualization of the Mandelbrot-set and has been used previously to compare hierarchical master-worker-skeletons [1].

On the main node the `master`-function gets streams from all of the `np` workers that are combined into a list using the `Lister`-construct. This list is split into results and worker ids and the latter are used as requests that are used to distribute tasks to workers. The `prefetch` number determines how many tasks are initially distributed to the workers.

Edges from the master to the workers have a filter `toWorkerSelect i` that

Listing 1.5. The core Master-Worker-Skeleton with Grace

```
mwGrace :: forall t r. (Trans t, Trans r) ⇒
           Int → Int →      -- no. of workers, prefetch
           ([t] → [r]) →    -- worker function
           [t] → [[r]]      -- tasks, result lists

mwGrace np prefetch wf tasks
  =  fst $ start $ build (0, master) (number workers) edges
  where
    master :: Lister ([[(Int, r)]] → ([[r]], [(Int, t)]))
    master = lister (λxs → (map (map snd) xs,
                     zip (initReqs ++ map fst (merge xs)) tasks))
                    [np]

    initReqs = concat (replicate prefetch [0..np-1])

    -- general worker function
    worker :: Int → [t] → [(Int, r)]
    worker i ts = zip [i, i..] $ wf ts

    -- workers concrete with id
    workers :: [Function]
    workers = toFL [worker i | i ← [0..np-1]]

    -- edge definitions
    edges :: [Edge Int Int]
    edges = zipWith4 E [1..np] [0,0..] [1,1..] nothings
          ++ zipWith4 E [0,0..] [1..np] [1,1..]
                        [Just (toWorkerSelect i) | i ← [0..np-1]]

    toWorkerSelect :: Int → ([r], [(Int, t)]) → [t]
    toWorkerSelect i (_, xs) = map snd $ filter ((==i) . fst) xs
```

selects for each worker the tasks assigned to it.

The user supplied worker function `wf` is used by a `worker` to evaluate tasks. The results are tagged with the worker id which is used as request for a new task. The (not shown) Grace functions `toFL` and `number` construct a list of worker nodes by adding labels from 1 to `np` to the list of functions. (Note that the worker with id i is represented by a node with label $i+1$.)

For the runtime measurements we used two systems. System NOW is a network of 8 Linux-workstations connected via Fast Ethernet. System 8C is a Linux-System with a 8-core CPU with 16 GB RAM running at 2.5 GHz on which we started 8 instances of the program. We computed the Mandelbrot-set visualization in a resolution of 5000 x 5000 pixels. The tasks were the lines of the graphic, i.e. 5000 tasks were evaluated. The prefetch value was 60.

The runtime on NOW fluctuated highly between 30 and 90 seconds for both programs, probably due to external influences on the network which could not be used exclusively. The difference in runtime between the two versions is too small to be measured accurately under these circumstances.

Figures 1.2 and 1.3 show trace visualisations, i.e. activity profiles, of two runs whose runtimes do not differ too much. The master-worker system consists of the master and 7 worker processes which have been placed on different PEs, with the master process on PE 1. The profiles look quite similar. The only noticeable

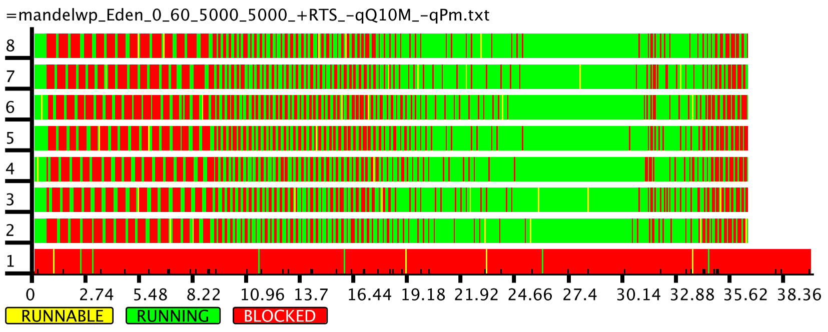

FIGURE 1.2. Trace of the Mandelbrot-Application on NOW, Version without Grace

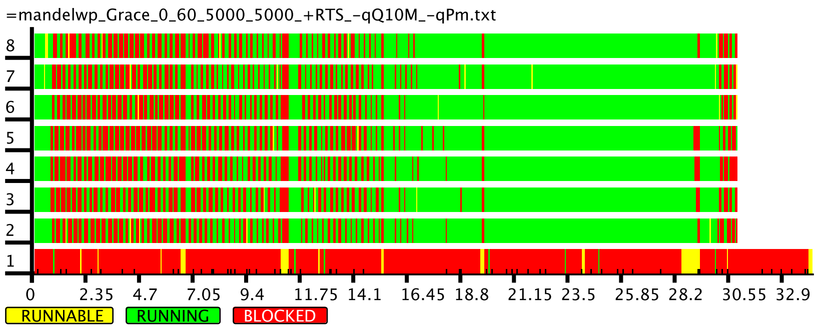

FIGURE 1.3. Trace of the Mandelbrot-Application on NOW, Version with Grace

difference lies in a higher amount of garbage collection (the yellow (light grey) sections) in the master process of the Grace version.

On 8C, where communication costs can be neglected, the traces show about the same characteristics. Only that no blocking can be perceived in the worker processes. We therefore omit these trace visualisations here. The higher amount of garbage collection in the master is apparent, too.

The runtimes on 8C are very stable. We measured the average of 5 runs. In addition to the runs with 5000 tasks we also measured the runtimes for bigger problem sizes. Table 1.1 gives the results. The runtimes for the three smaller problem sizes differ in approx. 0.2 seconds, twice in favor of the Grace-free version, once in favor of Grace. For 20.000 tasks the Grace program is suddenly 3 seconds slower. A trace reveals that this is due to a three-second garbage collection in the master process just one second before the end. The runtime differences of the bigger problem sizes are not as significant but can be ascribed to garbage collection in the master process, too.

It could be that the slightly higher amount of memory that is needed for

number of tasks / lines	5,000	10,000	15,000	20,000	25,000	30,000
runtime of version without Grace	25.87 s	51.94 s	77.11 s	102.59 s	127.69 s	153.53 s
runtime of version with Grace	26.08 s	51.72 s	77.34 s	105.71 s	128.40 s	154.62 s
time difference	0.21 s	−0.22 s	0.23 s	3.12 s	0.71 s	1.09 s

TABLE 1.1. Runtimes of the Mandelbrot-Application on 8C

the additional data structures used in the Grace version just happens to make garbage collection in the main process more often necessary. We suspect that some ressources remain in memory even after the process network has been built. A restructuring of the code for the main process may allow an earlier release of these ressources.

1.7 A MORE SOPHISTICATED EXAMPLE: HYPERQUICKSORT

While the master-worker example in the last section was a good test case for performance comparisons, it does not use the more powerful features of Grace due to the simplicity of the tree network. In this section, we give a more sophisticated example that uses all of Grace's features: an implementation of the hyperquicksort-algorithm [12], a parallel variant of the well known quicksort-Algorithm that works on a hypercube network.

A hypercube of dimension d consists of 2^d nodes, each of which is connected to d neighbours. The nodes can be enumerated from 0 to $2^d - 1$ in such a way, that a node p_i is connected to all nodes p_j where the binary representations of i and j differ in exactly one bit.

Algorithm Description

To sort a list of $n2^d$ elements the algorithm first distributes them evenly onto the available processor elements (PEs). The aim is to reorder them such that the sublists on every PE is sorted and for every pair of nodes p_i and p_j where $i < j$ the elements stored on p_i are smaller or equal to all elements stored on p_j.

To achieve this, every PE first sorts its local sublist using the sequential quicksort. Then, d phases follow. Let H_i^j denote the (sub)hypercube of dimension j that consists of the nodes $p_i, ..., p_{i+2^j-1}$. In any phase k, $k \in \{0, ..., d-1\}$, each hypercube H_i^{d-k} is split into the hypercubes H_i^{d-k-1} and $H_{i+2^{d-k-1}}^{d-k-1}$. For that, the root node of H_i^{d-k} first determines the median of its sublist and broadcasts it to the other nodes of H_i^{d-k}. Then, any node of H_i^{d-k-1} sends the upper part of his sublist with elements greater than the pivot element to his neighbour in $H_{i+2^{d-k-1}}^{d-k-1}$ and receives that one's elements smaller or equal to the pivot. Figure reffig:hqcomm exemplifies this. A merge of the newly received list with the list that remained on the node concludes the phase.

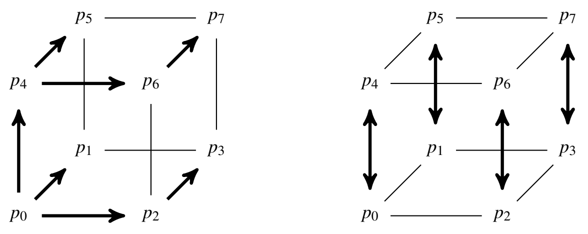

FIGURE 1.4. Communication in the first Phase of a 3D-Hyperquicksort: Broadcast of Pivot Element in H_0^3 (left); Exchange of sublists (right).

At the end of any phase k the elements in any hypercube H_i^{d-k-1} are smaller than those on H_j^{d-k-1} iff $i < j$. After the last phase, the original hypercube has been deconstructed down to subhypercubes with only one PE and the desired order is achieved.

Assuming a good selection of pivot elements the algorithm has an expected time complexity of $O(n \log n + \frac{d(d+1)}{2} + dn)$.

Realization

Listing 1.6 shows how the hypercube-network that sorts a given input list can be specified with Grace. The functional core is defined by the function `hypernode` of which we only give the type. Supplied with the nodes index and the initial sublist the curried function is placed on every node of the network. Additional parameters (for which we will define edges later) are a list of pivot elements and a list of sublists that are received in each phase of the algorithm. The result is a pair of two lists, one containing the pivot elements and one the sublists to be sent in each phase. The latter is supposed to include as last element the complete sublist a node has after the last phase. Note that the pivot element in each phase is either taken from the input list or computed from the stored elements, depending on whether the node is root of its subhypercube in a given phase or not.

The `Lister`-construct is used to assemble the `dim` sublists coming from different neighbours into a single list. The list of pivot elements on the other hand is received as a stream from only one other node. (Except for the root node with index 0 which computes all pivot elements by itself and for which this input is undefined.)

The `mainNode` not only computes the result of its `hypernode`-function but also the complete sorted list by receiving and concatenating all results of each node. This is another additional element of the list of computed sublists.

The edge definitions consist of three parts: Edges for the communication of the pivot element, for the exchange of sublists and a direct connection from each node to the root node that transfers the result lists.

The `pivotEdges` define a tree embedded into the hypercube. They are constructed using a helper function and carry a function that selects the pivot list

Listing 1.6. Definition of the Hyperquicksort-network

```
hyperquicksortNW :: forall a. (Ord a, Trans a) ⇒
                    Int → [a] → ProcessNetwork ([a], [[a]])
hyperquicksortNW dim input = build controller nodes edges
  where

    -- *** node definitions ***
    nodes = [N i (lister (hypernode i (inputs !! i)) [dim,0])
             | i  ← [1..2^dim -1]]

    mainNode :: (Int, Lister ([[a]] → [[a]] → ([a], [[a]])))
    mainNode = (0, lister mainFunction [2^dim - 1, dim])

    --              results  lists    (pivots, lists)
    mainFunction :: [[a]] → [[a]] → ([a], [[a]])
    mainFunction rs xs = (pivots, xs' ++ [last xs' ++ concat rs])
      where
        (pivots, xs') = (hypernode 0 (inputs !! 0)) xs undefined


    inputs = unshuffleN (2^dim) input


    --           nodeId input  lists    pivots  pivots, lists
    hypernode :: Int → [a] → [[a]] → [a] → ([a], [[a]])
    hypernode    nodeId es     lists    pivots = ...


    -- *** edge definitions ***
    edges = pivotEdges ++ dataEdges ++ resultEdges
      where

        -- edges for the broadcast of pivot elements
        pivotEdges = hlp (dim-1) [0]

        hlp (-1) _ = []
        hlp d ns = es ++ hlp (d-1) (ns++ms)
          where
            es = zipWith4 E ns ms (repeat dim)
                         (repeat (Just (fst :: ([a], [[a]]) → [a])))
            ms = map (+2^d) ns


        -- edges for exchange of sublists
        dataEdges = [E (complementBit i k) i (dim-k-1) (selectData k)
                     | i ← [0..2^dim - 1], k ← [0..dim-1]]

        selectData :: Int → Maybe (([a], [[a]]) → [a])
        selectData d = Just ((!!(dim-d-1)) . snd)


        -- edges for the collection of the final result
        resultEdges = [E i 0 (i-(2^dim)) selectResult
                       | i ← [1..2^dim -1]]

        selectResult :: Maybe (([a], [[a]]) → [a])
        selectResult = Just ((!!dim) . snd)
```

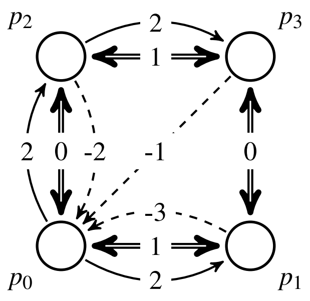

FIGURE 1.5. Topology of the Process Network for the Hyperquicksort in a 2D-Hypercube. Edges for the Communication of the Pivot Elements (single line), the Sublists (double line) and the Collection of the Result (dashed line). The Edge Weights define an Ordering on each Node's incoming Edges.

Listing 1.7. Definition of the `hyperquicksort`-function

```
hyperquicksort :: (Ord a, Trans a) ⇒
                  Int → [a] → [a]
hyperquicksort dim xs = (!!(dim+1)).snd $ start
                          $ hyperquicksortNW dim xs
```

from the result of the `hypernode`-computation. Since the pivot-list parameter of `hypernode` succeeds the sublists-parameter, the weight of the pivot edges is chosen to be `dim` which is greater than that of any data edge.

The `dataEdges` have weights of 0 to `dim`−1, corresponding to the phase in which they are used. Figure 1.7 shows the graph used in a hyperquicksort with a twodimensional hypercube. The `k` used in the list comprehension defines the size of the subhypercube that is to be split in a phase $d-k-1$. The `selectData`-function extracts the sublist to be transmitted from the result of `hypernode`.

Finally, after all phases a node's result is transferred via a direct connection to the `mainNode`. The ordering with negative weights was chosen such that it does not interfere with the other edges' weights and it defines the correct ordering on the received sublists in the main node which will only have to apply `concat` to get the overall result.

To complete the definition of the hyperquicksort, all that is left is to instantiate the network and select the desired reults from the networks computation (cf. listing 1.7).

1.8 RELATED WORK

The idea of splitting a parallel program into parts of sequential computations and a coordination definition is not new.

S-Net [2] is a coordination language targetted at stream processing that allows to specify the coordination network without too much knowledge of the connected computation *boxes*, which therefore can be implemenented in an arbitrary box language.

Even if we describe a computation as graph we have not much in common with visual languages or term-graphs [10] in general since we use the graph only as static specification and do not transform it.

1.9 CONCLUSION

Modeling process networks as graphs is more intuitive than the use of dynamic channels on a low level of abstraction. We have given a library that introduces these graph definitions into Eden with only a slight impact on the runtime of the written program which may even be reduced further. Currently, type safety is unfortunately not guaranteed but the unsafety is limited to communication interfaces between the processes. We hope to address this problem in the future.

Our modeling of computations as graphs is not limited to the parallel world. The same could be done sequentially. With only minor modifications we would obtain a tool to evaluate term-graphs.

REFERENCES

[1] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical Master-Worker Skeletons. In P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages*, LNCS 4902, pages 248 – 264. Springer, 2008. submitted for publication.

[2] C. Grelck, S.-B. Scholz, and A. Shafarenko. S-Net: A typed stream processing language. In *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary*, Technical Report 2006-S01, pages 81–97, 2006.

[3] O. Kiselyov. How to write an instance for not-a-function. `http://okmij.org/ftp/Haskell/typecast.html#is-function-type`. Online, 25.11.2008.

[4] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. Technical Report SEN-E0420, CWI, Amsterdam, Aug. 2004.

[5] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[6] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[7] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[8] S. Peyton Jones and J. H. (editors). Haskell 98: A non-strict, purely functional language. Technical report, Februar 1999.

[9] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[10] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice*. John Wiley and Sons Ltd., Chichester, UK, 1993.

[11] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.

[12] B. Wagar. Hyperquicksort - a fast sorting algorithm for hypercubes. In M. T. Heath, editor, *Hypercube Multiprocessors 1987 (Proceedings of the Second Conference on Hypercube Multiprocessors)*, pages 292–299. SIAM, 1986.