Reasoning About Skeletons in Eden

Ricardo Peña[a], Clara María Segura[a]

[a]Departamento Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

1. Introduction

The parallel-functional language Eden extends the lazy functional language Haskell by constructs to explicitly define and launch processes. Skeletons can be implemented in the language itself and can be invoked in parallel applications as higher-order functions. So, Eden can be used both as a *system-level* implementation language and as an *application-level* parallel one. In order to be useful, skeletons should be proved correct and should lead to an efficient use of the underlying machine.

The paper presents examples of system-level programming in Eden, showing the conciseness of defining skeletons at a high level of abstraction. Efficiency reasoning is provided by accurate cost models and correctness is proved by induction proofs. If the parallel program uses finite data structures, the proof implies also successful *termination*. If it uses infinite ones (i.e. streams), the proof implies *productivity*. A program is productive if, whenever it receives continuous input, it provides continuous output. Productivity amounts to deadlock freedom. The paper summarizes the correctness and efficiency results contained in some other papers, specially [5–8] and [4].

The plan is as follows: in Section 2, we briefly describe Eden's syntax and semantics; Section 3 shows examples of system level programming by defining two parallel implementations of the skeleton `map`, providing their respective cost models and proving these implementations correct; Section 4 survey other Eden skeletons in less detail. Finally, Section 5 provides some conclusions and related work.

2. Eden features summary

A *process abstraction* is just the application of the predefined function `process` to a function. If f id of type `a -> b` then `process f` is of type `Process a b`. It defines the behaviour of a process receiving `x::a` as input and returning `f x::b` as output. A *process instantiation* uses the predefined infix operator:

```
(#) :: (Transmissible a, Transmissible  b) => Process a b -> a -> b
```

In order to be able to transmit a value, its type must belong to the type class `Transmissible`. The evaluation of an expression `(process f) # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending `e2` via an implicitly generated channel, while the new *child process* will evaluate the application `f e2` and return the result via another implicitly generated channel. For input or output tuples, independent concurrent threads and channels are created to evaluate each tuple component.

Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access input not yet available, are temporarily suspended. This is the only way in which Eden processes synchronize. Let us remark that there are no explicit instructions handling channels or messages as communication/synchronization is completely implicit.

Replacing in the definition of `map`

```
map       :: (a -> b) -> [a] -> [b]
map f xs  =  [f x | x <- xs]
```

the function application `f x` by a process instantiation, leads to a simple parallel `map` skeleton in which a different process is launched for each element of the input list:

```
map_par :: (Transmissible a, Transmissible b) => (a -> b) -> [a] -> [b]
map_par f xs =  [process f # x | x <- xs] `using` spine
```

The `spine` strategy (see [11]) is used to eagerly evaluate the spine of the process instantiation list. Otherwise, the laziness of Haskell would prevent that all processes were immediately created.

Many-to-one communication is an essential feature for some parallel applications, but it spoils the purity of functional languages, as it introduces non-determinism. In Eden, the predefined process abstraction

```
merge :: Transmissible a => Process [[a]] [a]
```

is used to instantiate a process which fairly merges a list of input streams into a single (non-deterministic) output stream. The incoming values are passed to the output stream in the order in which they arrive. In this way `merge` provides many-to-one reactive communication. It can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. Even though the skeletons presented are deterministic, some of them are required to immediately react to requests for work coming from a group of worker processes. An instantiation of `merge` will propagate these requests as they are being produced.

Eden's compiler[1] has been developed by extending the Glasgow Haskell Compiler (GHC) [9]. Eden's runtime system (RTS) is an implementation of the DREAM abstract machine [1] on top of a message passing library. Both PVM and MPI can be used. Therefore, the compiler can be ported to any architecture where GHC and either PVM or MPI are available.

Eden provides no *placement annotations*. However, Eden's RTS supports two modes to map processes to processors, which can be chosen by the user for each execution. *Round-robin mode*: If several processes are instantiated from a particular processor p, they are mapped to consecutive processors starting with the one numbered one more than p. *Random mode*: Each processor maps instantiated processes to randomly chosen processors.

The number of processors is provided by the integer constant `noPe`. It can be used to adapt the number of processes to the number of available processors.

3. The `map` skeleton in Eden

The definition of a skeleton consists of two parts: (1) A *specification* giving the type of the skeleton and a description of its observable behavior as a higher-order function; and (2) the *implementations*. A skeleton may have several implementations, and for each one two pieces must be provided: A parallel program (the *algorithm*), and a formula describing its expected parallel execution time (the *cost model*).

Cost models describe the parallel time of the algorithm. For a survey of cost models see e.g. [2]. The cost models presented in this section are an adaptation to Eden of classical cost models

[1] Available at http://www.mathematik.uni-marburg.de/inf/eden

Problem dependent parameters	
N	Size of the input
t_f	sequential CPU time for function f
nwI	number of words of input message going to a child
nwO	number of words of output message coming from a child
RTS dependent parameters	
t_{create}	CPU time in a parent processor to create a child process
$t_{\#}$	CPU time in a child processor to create a new process
Architecture dependent parameters	
P	Number of processors
δ	latency of a message, from start sending to start receiving
λ	start-up fixed CPU cost for sending or receiving a message
β	per-word CPU cost for sending or receiving a message

Figure 1. Parameters of the cost models

appearing in the literature. In Figure 1 we show the different parameters involved in Eden skeletons cost models. We will use the abbreviations:

$$t_{unpackI} = t_{packI} = \lambda + \beta nwI$$
$$t_{unpackO} = t_{packO} = \lambda + \beta nwO$$

In the rest of this section we concentrate on two different implementations of the skeleton map. For each one we present its cost model and a proof of its correctness.

For proving correctness, we will use induction on natural numbers and $P(n)$ will denote a predicate where the natural n will be related to the length of one or more lists in the program being verified. Our aim is to prove $\forall n.P(n)$ using the ordinary induction rule:

$$\frac{P(0) \quad \forall n \geq 0.P(n) \Rightarrow P(n+1)}{\forall n.P(n)}$$

When the rule is applied to a function on finite lists, it proves the total correctness of the function, i.e. the proof implies termination. When the rule is applied to a recursively defined list, it proves that its length increases at each recursive call, i.e. the list is potentially infinite, and then the function producing it is productive.

In what follows, $|xs|$ denotes the length of the list xs and xs_i the ith element of the list xs starting from 0. For instance, the specification of `map` can be done by the following predicate:

$$P_{map}(n) \stackrel{\text{def}}{=} \forall f.\forall xs.\ |xs| = n \Rightarrow \texttt{map f } xs = [f\ x_i \mid x_i \leftarrow xs]$$

It is trivial to prove its correctness by using the recursive definition of `map`. Notice that the proof implies termination of `map` for finite lists xs and implies productivity for infinite ones.

3.1. Farm Implementation

In the farm implementation of `map`, called `map_farm`, a single process is created in each available processor and tasks are evenly distributed into processors. Funtion f is applied to each task and results are collected by the parent process. If the parent process load is low, we locate it in the same processor as one of its children. Otherwise, we devote a separate process for the parent. A threshold parameter is used to allow the skeleton to take this decision. This implementation is appropriate when task granularity is uniform and an even distribution of the number of tasks amongst all the processors is desired. The length of the task list must be rather higher than the number of available

processors in order to improve the load balance. In order to instantiate processes correctly, the round-robin mode is used by the RTS. The distribution and collection functions are also parameters of the skeleton. Here is the implementation:

```
map_farm :: (Transmissible a,Transmissible b) => Int -> (a -> b) -> [a] -> [b]
map_farm thr = farm np unshuffle shuffle  where np | noPe > thr = noPe - 1
                                                   | otherwise  = noPe
farm :: (Transmissible a, Transmissible b) =>
        Int -> (Int -> [a] -> [[a]]) ->([[b]] -> [b]) -> (a -> b) -> [a] -> [b]
farm np unshuffle shuffle f tasks = shuffle (map_par (map f) (unshuffle np tasks))
```

Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle . unshuffle n) xs == xs` holds. As this is standard Haskell programming, we do not show these functions here. The cost model for `map_farm` is the following:

$$\begin{array}{lcl} t_{map\_farm} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packI} + t_{unshuffle_1}) + \delta \\ L_{final} & = & \delta + t_{unpackO} + t_{shuffle_1} \\ t_{worker} & = & t_{\#} + \lceil \frac{N}{P} \rceil (t_{unpackI} + t_f + t_{packO}) \end{array}$$

In essence, $t_{map\_farm} = k_1 P + k_2 \lceil \frac{N}{P} \rceil + k_3$ for some constants k_1, k_2 and k_3. In [5], some actual executions are shown accurately agreing with this model.

In order to proof the correctnes of this implementation, we need to prove:

$$P_{map\_farm}(n) \stackrel{\text{def}}{=} \forall f. \forall \mathtt{xs}.\ |\mathtt{xs}| = n \Rightarrow \mathtt{map\_farm\ f\ xs} = [f\ x_i \mid x_i \leftarrow \mathtt{xs}]$$

We assume that the following predicate has been proved by induction on n:

$$P_{unshuffle}(n, np) \stackrel{\text{def}}{=} \forall n. \forall np. \forall xs.\ |\mathtt{xs}| = n \Rightarrow \mathtt{unshuffle\ np\ xs} = \mathtt{xss} \mid concat\ xss \in perm(xs)$$

where $perm$ gives the set of permutations of a list. We also assume that `map_par` satisfies the predicate $P_{\mathtt{map}}(n)$ of `map`. Then, we have the following equivalences:

$$\begin{array}{ll} & \mathtt{map\_par}\ (\mathtt{map}\ f)\ (\mathtt{unshuffle\ np\ xs}) \\ = & \quad \{\text{by the correctness of } \mathtt{map\_par}\} \\ & [\mathtt{map}\ f\ \mathtt{ys} \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the correctness of } \mathtt{map}\} \\ & [[f\ y \mid y \leftarrow \mathtt{ys}] \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the correctness of } \mathtt{map}\} \\ & [[z \mid z \leftarrow \mathtt{map\ f\ ys}] \mid \mathtt{ys} \leftarrow \mathtt{unshuffle\ np\ xs}] \\ = & \quad \{\text{by the predicate } P_{unshuffle}(n, np)\} \\ & [[z \mid z \leftarrow \mathtt{zs}] \mid \mathtt{zs} \leftarrow \mathtt{unshuffle\ np\ (map\ f\ xs)}] \\ = & \quad \{\text{by change of notation}\} \\ & \mathtt{unshuffle\ np\ (map\ f\ xs)} \end{array}$$

Then, by using the property `(shuffle . unshuffle np) xs == xs` we finally have:

$$\begin{array}{ll} & \mathtt{map\_farm\ f\ xs} \\ = & \mathtt{farm\ np\ unshuffle\ shuffle\ f\ xs} \\ = & \mathtt{shuffle\ (map\_par\ (map\ f)\ (unshuffle\ np\ xs))} \\ = & \mathtt{shuffle\ (unshuffle\ np\ (map\ f\ xs))} \\ = & \mathtt{(shuffle\ .\ unshuffle\ np)\ (map\ f\ xs)} \\ = & \mathtt{map\ f\ xs} \end{array}$$

3.2. Replicated Workers Implementation

The load balance obtained using the farm scheme can be poor when the granularity of the tasks is not uniform. Instead of using a fixed task distribution scheme, we can distribute work on demand.

This gives rise to the *replicated workers* implementation of `map`. Initially, the manager assigns two tasks to each of the workers. By assigning more than one task, the idle time between tasks is minimized. Each time a worker finishes a task, it sends an acknowledgment message to the manager including the task result, and then a new task is assigned to that process. The computation finishes when the manager has received all the task results. By using the process `merge`, acknowledgments from different processes can be received by the manager as soon as they are produced. If each acknowledgment contains the identity of the sender process, the list of merged results can be inspected in order to know who has sent the message, and then a new work can be assigned to it. In Eden, this solution can be expressed as a set of mutually recursive list definitions:

```
map_rw f ts = let tids = zip [0..] ts in
 letrec  outs =  let urs = merge # outs
                     reqs= [0..np-1] ++ [0..np-1] ++ map first urs
                     ins = distribute tids reqs
                 in  [(worker f i) # in | (i,in) <- zip [0..] ins]
 in sortMerge outs

worker f i =  process (\ts -> map (\(it,t) -> (i,it,f t)) ts)
```

The list `tids` just assign a different number to each task. The list of lists `outs` contains a result list coming from each worker. The list `urs` produced by `merge` can be understood as the temporal sequence of requests for new tasks. Function `distribute` assigns a new task form the list `tids` to the worker who has first sent a request. The implementation of `distribute` and `sortMerge` are not shown. They are assumed to satisfy the following predicates:

$$P_{distribute} \stackrel{\text{def}}{=} |xs| \leq |rs| \wedge \forall i \in \{0..\,|rs|-1\}.rs_i \in \{0..np-1\} \Rightarrow$$
$$\texttt{distribute xs rs} = \texttt{yss} \mid \; |yss| = np \wedge (\forall j \in \{0..np-1\}.ordered(yss_j))$$
$$\wedge \textstyle\sum_{i=0}^{np-1} |ys_i| = |xs| \wedge concat\ ys \in perm(xs)$$
$$P_{sortMerge} \stackrel{\text{def}}{=} \texttt{sortMerge xss} = \texttt{rs} \mid rs = map\ third\ xs \wedge ordered(xs) \wedge xs \in perm(concat\ xss)$$

That is, `distribute` behaves rather similarly to `unshuffle` and `sortMerge` produces an ordered list from a list of ordered lists. If the number N of tasks is much greater than the number P of processors, the cost model for `map_rw` is:

$$\begin{array}{lcl} t_{map\_rw} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packI} + t_{distribute_1}) + \delta \\ L_{final} & = & \delta + t_{unpackO} + t_{sortMerge_1} \\ t_{worker} & = & t_{\#} + \frac{N}{P}(t_{unpackI} + t_{comp} + t_{packO}) \end{array}$$

Assuming a perfect load balance, it can be considered that every worker receives the exact average number of tasks, each task costing the average computing cost $t_{comp} = \frac{1}{N}\sum_{i=1}^{N} t_{f_i}$, where t_{f_i} represents the cost of function f when applied to task i. In $t_{distribute_1}$ we consider accumulated the costs of `zip, ++` and `map first` functions when producing one element. Notice that the ceiling operation has disappeared from $\frac{N}{P}$. Then, the simplified model is $t_{map\_rw} = k_1 P + k_2 \frac{N}{P} + k_3$. This is very similar to that of t_{map_farm} but now tasks of different granularities are allowed. Again, [5] shows examples accurately agreeing with this model.

In order to prove the correctness of the skeleton the following predicates are proposed for each auxiliary list produced by the algorithm:

$$P_{outs}(ts, f, n) \stackrel{\text{def}}{=} \exists n_0 \ldots n_{np-1}.n = \sum_{i=0}^{np-1} n_i \wedge \forall i \in \{0..np-1\}.n_i = |outs_i| \wedge ordered(outs_i) \\ \wedge (\forall j \in \{0..n_i - 1\}.r = f\ ts_{it} \wedge iw \in \{0..np-1\} \quad where\ \ (iw, it, r) = (outs_i)_j\ )$$

$$P_{urs}(n) \stackrel{\text{def}}{=} \forall i \in \{0..n-1\}.iw \in \{0..np-1\} \quad where\ \ (iw, \_, \_) = urs_i$$

$$P_{reqs}(n) \stackrel{\text{def}}{=} \forall i \in \{0..n-1\}.reqs_i \in \{0..np-1\}$$

$$P_{ins}(ts, n) \stackrel{\text{def}}{=} \exists n_0 \ldots n_{np-1}.n = \sum_{i=0}^{np-1} n_i \wedge \forall i \in \{0..np-1\}.n_i = |ins_i| \wedge ordered(ins_i) \\ \wedge (\forall j \in \{0..n_i - 1\}.reqs_{it} = i \wedge t = ts_{it} \quad where\ (it, t) = (ins_i)_j\ )$$

The first one expresses that `outs` consists of np lists whose lengths sum is the number n of tasks. Each sublist consists of triples (iw, it, r) and is ordered by task identity it. The worker identities iw are numbers in the range $0..np-1$, and values r are the result of applying f to the task in position it of list ts. The only important property of the second predicate is that the first component of each triple of `urs` is an actual worker identity in the range $0..np-1$. A similar comment applies to `reqs`. The last predicate expresses that `ins` consists of np lists whose lengths sum is n. Each sublist consists of tuples (it, t) and is ordered by task identity it. It also says that tasks are located in `ins` according to the requests in list `reqs`. The main theorem to be proved is that `map_rw` behaves as `map`:

$$P_{map\_rw}(n) \stackrel{\text{def}}{=} \forall f. \forall xs.\ |xs| = n \Rightarrow \texttt{map\_rw f}\ xs = [f\ x_i \mid x_i \leftarrow xs]$$

We first prove the following predicate $Q(n)$ by induction on the length n of the task list:

$$Q(n) \stackrel{\text{def}}{=} \forall f. \forall ts.\ |ts| = n \Rightarrow P_{outs}(ts, f, n)$$

The proof scheme is as follows:

Base case $n = 0 \Rightarrow ts = [\,] \Rightarrow P_{outs}([\,], f, 0)$

Inductive step Assuming a task list ts of length $n+1$ and the induction hypothesis $P_{outs}(ts', f, n)$ for a list ts', the prefix of lenght n of ts, we easily prove from the program:

$$P_{outs}(ts', f, n) \Rightarrow P_{urs}(n) \Rightarrow P_{reqs}(n + 2np)$$

Assuming $P_{reqs}(n + 2np)$, $P_{distribute}(tids, reqs)$ and $|tids| = n + 1$, by the definition in the program of `ins`, `outs` and `worker`, we have:

$$P_{reqs}(n + 2np) \wedge |tids| = n + 1 \Rightarrow P_{ins}(ts, n + 1) \Rightarrow P_{outs}(ts, f, n + 1)$$

and then we are done.

This amounts to proving $\forall n.Q(n)$. Knowing that `map_rw f ts = sortMerge outs` and assuming $P_{sortMerge}(outs)$, it is straightforward to prove $\forall n.P_{map\_rw}(n)$, i.e. `map_rw` terminates and is correct for all finite lists.

Productivity is an added value of some programs. It says that, even when the program does not terminate, it will produce useful output during its work by processing increasing portions of its infinite input. For instance, a function reversing a list is terminating for finite input but non

productive for an infinite one. Our `map_rw` skeleton produces an infinite result for an infinite number of tasks. For lack of space, we only point out some key ideas about the proof:

First, the predicates $P_{outs}(ts, f, n)$, $P_{distribute}$ and $P_{map\_rw}(n)$ must be slightly modified. The new definitions will allow that list `outs`, the output sublists produced by `distribute` and the list produced by `map_rw` may have any number of elements. Predicates $P_{distribute}$ and P_{map_rw} will have an additional parameter n indicating the number of tasks processed up to some point in time. Predicates $oredered$, $perm$ and $concat$ will be modified accordingly. The main theorem is now:

$$|ts| = \omega \Rightarrow \forall n. \forall f. P_{outs}(ts, f, n)$$

Again the proof is done by induction on n. The key step is to show that $P_{outs}(ts, f, n + 2np) \Rightarrow P_{outs}(ts, f, n+1)$, which can be done by reasoning that $[ts_0..ts_n]$ is a prefix of $[ts_0..ts_{n+2np-1}]$. This will show that the list `outs` is productive. By separately proving that `sortMerge` is productive, also will do `map_rw`. More details can be found in [6] and [8].

4. Other Eden Skeletons

More examples of Eden skeletons can be found in [5,4]. For instance, there is a *divide and conquer* skeleton where a dynamic tree of processes is created in which each process is connected to its parent. An integer parameter determines the maximum level after which no more children processes are generated, and the sequential version is used instead. The implementation is as follows:

```
dc_naive ::(Transmissible a, Transmissible b) =>
    Int -> (a -> Bool) -> (a -> b) ->( a -> [a]) -> (a -> [b] -> b) -> a -> b
dc_naive 0 trivial solve split combine  = dc trivial solve split combine
dc_naive d trivial solve split combine x
  | trivial x   = solve x
  | otherwise   = combine x c
 where c = map_par (dc_naive (d-1) trivial solve split combine) (split x)
```

Notice here that there is no single manager process, as it was the case in the `map` skeletons, because every child is a parent process of the next process level. A better implementation is obtained by first flattening the task tree and producing a list of tasks. Then the `map_rw` skeleton is used to solve the tasks and finally a single solution is computed from the list of results.

A pipeline skeleton consists of a list of stages. Each stage applies a different function to the result obtained in the previous one. A naïve parallelization of this scheme can be done by instantiating a different process to evaluate each of the pipeline stages. This can be expressed in Eden as follows:

```
pipe_naive :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipe_naive fs xs  = (ppipe fs) # xs

ppipe :: Transmissible a => [[a]->[a]] -> Process [a] [a]
ppipe [f]    = process f
ppipe (f:fs) = process (\xs -> (ppipe fs) # (f xs))
```

This definition does not achieve the desired topology because the last process of the pipe cannot sends the values directly to the main process. Instead, a hierarchical topology is created. In order to get a flat one, a better definition makes use of Eden's *dynamic channels* facility, not explained in this paper (see for instance [4]). In [7] a manual transformation was proposed which allows to obtain a flat solution based on dynamic channels by using as specification a hierachical one. The method has been also applied to ring, grid and torus skeletons. Cost models and proof of correctness have been produced for most of these implementations (confirm in [5,8]).

5. Conclusions

The main differences between Eden and more traditional skeleton-based languages are two: (1) Eden is functional while the vast majority of skeleton implementation languages are imperative, and (2) skeletons can be implemented and used within the same language. In other approaches, skeletons are often implemented in a low-level language different from the one in which they are used. For instance, in *PMLS* [10] Scaife et al. extend an ML compiler by machinery which automatically searches the given program for higher-order functions which are suitable for parallelisation. During compilation these are replaced by efficient low-level implementations written in C and MPI.

The main point of this paper has been showing that the conciseness of the functional notation allows to conduct formal proofs of the skeletons with a reasonable effort. Both equational and predicate based reasoning have been used, and both termination and productivity of skeletons have been shown by induction on natural numbers. In [8] Eden skeletons were proved terminating and/or productive by using an extension of the sized type theory by Hughes and Pareto [3].

Additionaly, cost models have been defined from the functional implementations in which low level runtime system an arquitecture parameters appear. These cost models allow the programmer to reason about the runtime behaviour of the corresponding parallel programs.

References

[1] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *IFL'97, Selected Papers. LNCS 1467*, pages 250–269. Springer-Verlag, 1998.

[2] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering. Heriot-Watt University, 2000.

[3] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.

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

[5] R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *Principles and Practice of Declarative Programming PPDP'01. Florencia (Italy). ACM Press*, pages 187–198, September 2001.

[6] R. Peña, F. Rubio, and C. Segura. Convenience, Efficiency and Correctness in the Parallel Functional Language Eden. In *Integrated Design and Process Technology, IDPT'02. Pasadena (EE.UU.)*, pages 1–10, June 2002.

[7] R. Peña, F. Rubio, and C. Segura. Deriving Non-Hierarchical Process Topologies. In *Trends in Functional Programming 3. Selected Papers of the 3rd Scottish Functional Programming Workshop, SFP'01. Intellect*, pages 51–62, 2002.

[8] R. Peña and C. Segura. Sized Types for Typing Eden Skeletons. In *Selected papers of Implementation of Functional Languages, IFL 2001*, pages 1–17. LNCS 2312. Springer, 2002.

[9] S. L. Peyton Jones. Compiling Haskell by Program Transformations: A Report from the Trenches. In *ESOP'96, LNCS 1058*, 1996.

[10] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative Cross-Platform Results from a Parallelizing SML Compiler. In *EuroPar 2001*, Manchester, England, 2001.

[11] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.