Channel Structures in the Parallel Functional Language Eden*

Silvia Breitinger, Rita Loogen
Philipps-Universität Marburg
{breiting,loogen}@mathematik.uni-marburg.de

Abstract

In this paper, we describe the handling of channel structures in Eden. Channel structures are crucial for achieving the flexibility and compositionality that is one of the most important aims of Eden. We show alternative mechanisms for treating them and explain why it is useful to admit the dynamic splitting of channel structures as a feature of the language. The modelling of this in the operational semantics as well as the smooth integration of channel structures into the parallel abstract machine DREAM proves the practicability of the approach[1].

1 Introduction

The parallel functional language Eden [6] is defined by adding a coordination language with explicit processes to the computation language Haskell. This coordination language in particular contains *process abstractions*, which start with the keyword `process` and define a scheme for a process in a functional style. *Process instantiations* are expressions that generate processes and yield the respective outputs as their result. A process abstraction is supplied with a tuple of input expressions using an operator `#`.
Upon process creation, communication channels for the transmission of data between parent and child will be established. *Communication channels* are unidirectional 1 : 1 connections which automatically transmit output expressions in normal form. Lists are transmitted in a *stream*-like fashion, i.e. piecemeal. An evaluation will be suspended if it depends on input that has not yet been received.
Dynamic reply channels can be used in order to establish new connections at runtime. If an expression `new` (*ch_name*, *chan*) *exp* is evaluated, a reference *ch_name* and a new input channel *chan* become visible in expression *exp*. The reference *ch_name* can be transmitted to a remote process which can then send the reply exp_1 to *ch_name* by *ch_name* `!*` exp_1 `par` exp_2. After this connection is built up, the computation of the sender continues with exp_2.
The evaluation of Eden processes is driven by the evaluation of the output expressions, for which there is always demand. This rule overrides normal lazy evaluation in favour

*Supported by the DAAD (Deutscher Akademischer Austauschdienst) and the Spanish Ministry of Education and Science in the context of the German-Spanish Acción Integrada n.142B.

[1] A more detailed internal report on the same topic is [4].

of parallelism. There will be a separate concurrent thread of execution for each output channel.

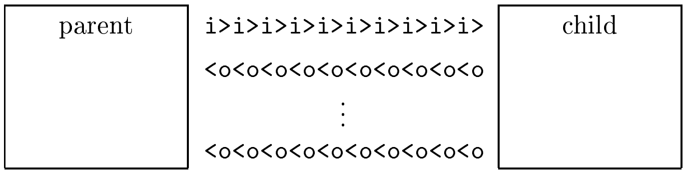
Two semantic levels are used in order to reflect the two level structure in the language. The lower level of the operational semantics describes local transitions inside of processes. Such transitions, which are denoted by the symbol $\vdash$, are connected to so-called *actions* of the system level. These actions give rise to global transitions, which are denoted by $\models$ and have an effect on the states of other processes. For example, if on the local level a process instantiation is evaluated, this is reflected by a **create** action on the system level. In addition, there exist actions for communication, closing of channels, generation and use of dynamic channels, and an ‘empty’ action for purely local process activities.

Arbitrary communication topologies can be created in Eden using only tuples of channels and dynamic reply channels. An illustrative example is shown in Section 2. But, in general, this approach is too cumbersome and low level and may lead to inefficient programs. In this paper we discuss a more abstract and legible method to define expressive communication interfaces.

Channel structures are data structures that comprise expressions as their components that have to be communicated via separate channels. This is a typical situation one finds with skeleton programs, see for example [7]. The representation of such structures by (structures with) *multiple communication channels* is crucial for programs that rely on a fair treatment of multiple components of a structured set of channels.

Example 1

Consider for instance a process `parent` that creates a process `child` which receives a (possibly infinite) stream of numbers as its input and produces a list of lists as its output. The child process uses a function `bucket` to distribute the input numbers (`i`) into `n` lists of output numbers (`o`), according to the remainder of their division by `n`.




```
parent :: Process  a b
parent = process input -> output
         where output = someComputation childOut
               childOut = child k # (produceIntList input)

child :: Int -> Process [Int] [[Int]]
child n = process inp -> bucket n inp
          where
          bucket :: Int -> [Int] -> [[Int]]
          bucket n [] = [ [] | k <- [0..(n-1)]]
          bucket n (l:list) = [ restLists !! k | k <- [0..(n-1)], k < r] ++
                              [ l : restLists !! r] ++
                              [ restLists !! k | k <- [0..(n-1)], k > r]
                              where
                                restLists = bucket n list
                                r = l `mod` n
```

The parent process has not enough information about the structure of the output `childOut` produced by the child process. It may be crucial that the input is supplied in the form of a *list of channels* and not in the form of *one* channel that transmits the whole input. Otherwise it would be impossible to access component number i if one of the components 1 to $i-1$ were infinite. Not only would the transmission in the form of one stream not work for infinite sublists, it would also lead to avoidable delays for finite ones. ◁

Given the above example, it is plausible that we can't do without some kind of channel structures completely. Not quite so obvious is the range of different structures that should be allowed. This also depends on the mechanism chosen to handle them.
In Section 2, various static vs. dynamic approaches to the handling of channel structures will be compared. Afterwards, in Section 3, we explain the formal treatment of the alternative selected and show some example evaluations. In order to give a broad perspective of the consequences of our choice, questions of implementation and complexity as well as possible optimizations are considered in Section 4. Finally, Section 5 presents related work and Section 6 gives an outlook on further work and draws conclusions.

2 Mechanisms for the handling of channel structures

In the previous section, we have motivated the support of structures of channels in Eden. In this section, we will have a closer look at the possible mechanisms that could be employed in order to provide such a language feature.
We proceed from the following hierarchy of language features for the handling of multiple in- / and outputs of the same type, with increasing flexibility.
If only tuples of channels were allowed, any change in the number of components would require major changes in the source program. The example sketched above has illustrated the need for processes with a flexible number of communication channels, so that this minimal approach is immediately ruled out.

Alternative 1: instantiation at *compile* time

If the language supported lists (or arrays) of channels with the number of elements fixed at *compile time*, a change in the number of components would require the changing of a parameter in the source program.
Despite its limitations, this alternative may be desirable from an implementation point of view, because the creation of a process and its channels then could be performed without any delay. This alternative will not be considered further in this paper because it is not well suited for a functional compiler and runtime system, where structures will not be allocated statically at compile time. It could only be used successfully in connection with special compile-time analysis steps.

Alternative 2: instantiation at *process creation* time

In the following we will analyze lists or arrays of channels, where the number of elements only has to be fixed at the time of the *instantiation* (of a process), like arrays in Haskell. Determining the number of channels at runtime, prior to process creation, is already much more flexible than alternative 1 and would suffice for a wide range of application programs.

This restriction however means that both input and output structures have to be instantiated at process creation time. Special syntax would be required to express the structure of an output and this would have to be used in connection with the process *instantiation* in the parent process. Note how artificial a restriction this would be: although the output is of course generated by the child and not by the parent, the parent has to contain a specification of the structure of this output. This would mean a burden for the programmer and an additional source of runtime errors, if the structure specified did not correspond to the one actually produced.

In the following, we will discuss a possible alternative, namely to construct channel lists indirectly. We will describe a transformation of the program in Example 1 that eliminates explicit lists of outports and introduces lists of dynamic reply channels instead. This emulation is interesting in its own right, because it clarifies the expressive power of Eden in its initial form and illustrates the use of dynamic channels.

Example 2

The following approach to the production of a list of channels uses mutual recursion between the two processes. The evaluation consists of four consecutive phases as indicated by the numbers in the diagram:

```
+-----------+                                    +-----------+
|  parent   |  >i>i>i>i> - 1 - >i>i>i>i>         |   child   |
|           |  <c<c<c<c< - 2 - <c<c<c<c<         |           |
|           |  >r>r>r>r> - 3 - >r>r>r>r>         |           |
|           |  <O<O<O<O< - 4 - <O<O<O<O<         |           |
|           |              .                     |           |
|           |              .                     |           |
|           |  <O<O<O<O< - 4 - <O<O<O<O<         |           |
+-----------+                                    +-----------+
```

1. The parent creates the child process with its channels `i` and `c` and supplies it with input `i`.
2. The child generates output `c`, which encodes the number of channels needed.
3. The parent generates output `r`, which contains the references to the requested reply channels.
4. The child establishes the `n` connections `O` and sends its output to the parent.

In the diagram, lower case letters are used for usual communication channels (produced upon process creation) and upper case letters for dynamic ones.

```
data Signal = S deriving (Eq, Show)

parent :: Process a b
parent = process input -> output
          where
          output = someComputation childOut
          ----- c -----              ----- r -----  --------- i --------
          ctrlFromChild = child k # (replyChanList,(produceIntList input))
                          --- O --
          (replyChanList, childOut) = provideChans ctrlFromChild
          provideChans :: [Signal] -> ([Chan_name], [[Int]])
          provideChans [] = ([],[])
```

```
          provideChans (S:rest) = new (cName, cCont)
                                  (cName:restNames, cCont:restCont)
                                  where    (restNames, restCont) = provideChans rest

child :: Int -> Process ([Chan_name],[Int]) [Signal]
child n = process (replyChans, inp) ->
          (requestChans findNumChan) ++ (useDynChans replyChans results)
          where
          findNumChan = getNum n inp -- number of channels needed
          requestChans :: Int -> [Signal]
          requestChans 0 = []
          requestChans n = S:(requestChans (n-1))

          useDynChans [] [] = []
          useDynChans (c:restChan) (o:restOut) = c !* o par
                                                 useDynChans restChan restOut
          results = bucket n inp
          bucket :: Int -> [Int] -> [[Int]]   -- as before
```

The child generates a *list* of signals `ctrlFromChild`: Not only can the parent already start to generate dynamic channels even if the total number of channels needed is not known yet, this solution creates demand by appending the empty result of useDynChans to its output channel `ctrlFromChild`. Here, a dependency has been introduced artificially in order to create demand for the threads that communicate with the dynamic channels.

In summary, the following had to be changed:

* In `child`, the knowledge about the number of output channels of the child had to be made explicit in the definition of identifier `findNumChan`.

- The 'real' output of `child` had to be removed from the interface and its transmission had to be hidden in the function `useDynChans`.

- A new output of `child` had to be introduced: a list of signals, one per required channel. It is generated by `requestChans` and the empty result of `useDynChans` is appended to it in order to create demand.

- The parent has to use a function `provideChans` that consumes these signals from the child and generates a pair of results: the list of channel names to be sent to the child and the list of channel contents to be consumed by `someComputation`.

If the programmer specified information about the number of channels explicitly (item * above), the computations in the process bodies could be wrapped up automatically by functions in the style of `requestChans` and `useDynChans` on the side of the sender of the channel list and `provideChans` on the side of the receiver. ◁

But even if this approach could easily be automated in this way, nonetheless this would not be the method of choice. Such a transformation may remove the expense for the *programmer*, but the *runtime* cost of such a solution would still be considerable. The subsequent alternatives will make use of mechanisms that supersede this requirement. Our success in emulating channel lists by reply channels already indicated that the mechanisms needed to allow channel lists are more or less at hand.

Alternative 3: dynamic splitting of channel *lists*

We will now generalize the language so that *lists* of channels can evolve arbitrarily during runtime, like lists in Haskell. Only the interface to the source language has to be designed carefully in order to keep up the ease of programming that the existing features of Eden offer. So it must be clarified in programs like the one in Example 1 whether a list of lists should be handled as a stream of lists or as a list of streams. For that purpose annotations `<.>` can be introduced in the type declaration of process abstractions in order to mark channel types. Thus, the type `[<a>]` defines a list of channels while `<[a]>` denotes a single stream channel. The annotations do not change the types of expressions. When evaluating a list of channels the sender process splits it up into a channel for the head of the list and a channel structure for the tail of the list. Channel lists are transparent in the sense that processes with a flexible number of outports can be programmed exactly as if they produced an ordinary list of outputs. The body of the child process in Example 1 need not be changed.

Alternative 4: dynamic splitting of channel *structures*

As the most powerful alternative, we will now discuss the dynamic handling of arbitrary data structures of channels in the style of the extensions shown in the previous subsection. Of course, it may be argued that the effects possible with general channel structures can be achieved with lists alone. Note that for message passing communication between two processes, there will be no real difference between a list with n channels and a tree that contains n channels. The only difference between these two versions is: who has to take care of the taking apart and putting together of the n elements: the system or the programmer.
Nevertheless the introduction of general channel structures can increase the comfort considerably, because the transmission of channel structures doesn't have to be handled explicitly on the level of the source program. This will be illustrated by the following example:

Example 3

Consider a large data base organized as a search tree and the problem of searching this data base for information that is not accessible via the indexes of the search tree. In the following program a simple search engine is defined as a process which uses separate threads to answer requests. Each thread processes the information in a node of the data base. The whole process outputs a tree of channels with the same shape as the data base tree.

```
data Tree inf  =  Leaf inf  | Node inf (Tree inf) (Tree inf)

searchEngine:: (Tree inf) -> Process  [req]       (Tree <[ans]>)
searchEngine   dataBase   =  process  requests -> resultDataBase
                             where
                               resultDataBase = mapT (handle requests) dataBase
                               mapT f (Leaf a)     = Leaf (f a)
                               mapT f (Node a l r) = Node (f a) (mapT f l) (mapT f r)
                               handle :: [req] -> inf -> [ans]
```

The single output channels can independently be accessed by the environment and directed to their real consumers. Artificial synchronisation constraints in the transfer of structured information are removed. The approach is more type-safe, as the communication structure need not be flattened into lists and immediately reflects the structure of the interface. $\triangleleft$

The above discussion has shown that alternative 4 is the alternative of choice. We will refer to this alternative as 'dynamic handling' in the following. In the next two sections, we will show syntax, semantics and implementation of this new language concept in detail.

3 Dynamic handling of channel structures

3.1 Formal treatment

In this section, we will introduce the syntactical and semantical prerequisites that are needed to support the channel structures of alternative 4.

By use of the channel annotations `< >`, the user can define arbitrary data structures of communication channels. The annotations were invented in order to be able to select the type of the individual communication channel(s): the types that are enclosed by the annotations will be the types of the data items transmitted.

Formally, we distinguish between message and interface types. The set of *message types* *MType* contains all datatypes that can be used as types of messages in Eden programs. The set *IType* of *interface types* contains all datatypes that can occur as types of communication channels or channel structures. The definition of these sets is mutually recursive. Let *TVar* be a set of type variables and $\Theta = \cup_{i=0}^{\infty} \Theta^i$ be a ranked alphabet of type constructors. Then, *MType* and *IType* are inductively defined as the smallest sets that contain the following:

- $TVar \subseteq MType$
- $\sigma \in \Theta^n, t_1, \ldots, t_n \in MType$
  $\Rightarrow \ (\sigma\ t_1 \ldots t_n) \in MType$
- $t_1, t_2 \in MType \Rightarrow \ t_1 \rightarrow t_2 \in MType$
- $t_{11}, \ldots, t_{1m}, t_{21}, \ldots, t_{2n} \in IType,\ m \geq 0,\ n \geq 1$
  $\Rightarrow \mathbf{Process}(t_{11}, \ldots, t_{1m})(t_{21}, \ldots, t_{2n}) \in MType$

- $MType \subseteq IType$
- $t \in MType \ \Rightarrow \ <t> \in IType$
- $\sigma \in \Theta^n, t_1, \ldots, t_n \in IType$
  $\Rightarrow \ (\sigma\ t_1 \ldots t_n) \in IType$

IType is a superset of *MType*, because any type is admitted for transmission on channels. Channel annotations are necessary to distinguish between channels of structures and structures with channels. Nested occurrences of annotations make no sense except for channels that carry process abstractions.

In Example 1, annotations must be introduced in the output specification of the process `child :: Int -> Process [Int] [<[Int]>]` to express that the output of type `[[Int]]` has to be transmitted in the form of a *list of channels*.

In order to clarify the complete structure of process interfaces, we introduce a *normal form of interface types* in which all channel types are marked with annotations. The function *itype_nf* defines the normal form representation for the elements of *IType*:

$$
\begin{array}{lcll}
\mathit{itype\_nf}\ (\ t\ ) & = & \texttt{<}t\texttt{>}\ \text{ for } t \in \mathit{MType} & \\
\mathit{itype\_nf}\ (\texttt{<}t\texttt{>}) & = & \texttt{<}t\texttt{>} & \\
\mathit{itype\_nf}\ (\sigma\ t_1 \ldots t_n) & = & (\sigma\ \mathit{itype\_nf}(t_1) \ldots \mathit{itype\_nf}(t_n)), & \text{if } \ (\sigma\ t_1 \ldots t_n) \notin \mathit{MType} \\
 & & & \text{and } \ \sigma \in \Theta^n
\end{array}
$$

It is applied to all component types in the type of process abstractions to derive the normal form of the latter. The *normalization of the interface* of process abstraction `child` yields `Int ->` **Process** `<[Int]>  [<[Int]>]`. This means that the initial interface of the child process will consist of one regular channel for input and a channel structure of type list for output.

Eden's operational semantics comprises two levels of transition systems. On the upper level global effects on process systems are described. The lower level handles local effects within processes. The interface between the two levels is formed by so-called 'actions' which communicate the need for global events to the upper level. There are actions for input from channels, output on channels, reply channel generation, process creation and process termination. The 'empty' action τ represents internal evaluations.

The local behaviour of a process p describes its own evolution within the whole system and is modelled by transitions on the set *Process* of processes: $\vdash\, \subset \mathit{Process} \times \mathit{Act} \times \mathit{Process}$. A local transition may be connected with a global transition, which is expressed by the presence of a non-empty action. An element of the set *Process* consists of its id, its input- and output interfaces (names of the active communication ports), and its local environment: $\mathit{Process} = \mathit{Process\_Id} \times \mathit{Channel\_Id}^2 \times \mathit{Script}$

The global behaviour of process networks is defined by a transition relation on system configurations, i.e. sets of *processes* and *channels*: $\models\, \subseteq \mathit{System} \times \mathit{System}$, where $\mathit{System} = \mathcal{P}(\mathit{Process} \cup \mathit{Channel})$, and $\mathit{Channel} = \mathit{Channel\_Id} \times \mathit{Channel\_Tag} \times \mathit{Process\_Id}^2 \times \mathit{Contents}$. This means, each channel is represented by a unique identifier, a tag that encodes the type of the contents, the process identifiers of its sender and its receiver, and the contents of the channel buffer.

In order to model the dynamic handling of channel structures, the operational semantics has to be extended by a representation of channel structures and an action **split**: *Channel structures* are represented by channels that have as channel tag a *type* contained in *IType*. The action (**split** $s, t, \ldots$) means that the channel structure s is split up into a term structure t with channels and/or channel structures as its components. It contains some additional information $(\ldots)$ for optimizations which will not be discussed here (for details see [5]). On the process level a **split**-action is initiated whenever a process evaluates a channel structure. The following rule describes the local transition in this case. In this rule, a simplified tag **ch** for channels is used, which can be instantiated either with **oneVal** or with **stream**.

$$
\begin{array}{|l|}
\hline
\langle p_{send},\ In,\ \ Out \oplus \{(o^{struct}, type)\}, \qquad\qquad\qquad eqs \oplus \{o^{struct} = C\ x_1 \ldots x_k\}\rangle \\
\vdash_{(\mathbf{split}, (o^{struct}, type),\ C\ o_1 \ldots o_k,\ \ \ldots)} \\
\langle p_{send},\ In,\ \ Out \oplus \{(o^{struct}, type), o_1, \ldots, o_k\},\ \ eqs \oplus \{c_1 = x_1, \ldots, c_k = x_k\}\rangle, \\
\text{where} \qquad t_1 \to \ldots \to t_k \to type \text{ is an appropriate type instance of } C, \\
\qquad\qquad\quad c_j \ \text{ are new globally unique channel identifiers,} \\
\text{and} \qquad\quad o_j = \begin{cases} (c_j, \mathbf{ch}) & \text{if } t_j \in \{< t >|\ t \in MType\} \\ (c_j, t_j) & otherwise \end{cases} \quad (1 \leq j \leq k) \\
\hline
\end{array}
$$

Whenever the output interface *Out* of a process contains a channel structure $(o^{struct}, type)$ and the local environment of the process contains an equation $o^{struct} = C\ x_1 \ldots x_k$ defining this structure, a **split**-action is initiated. The channel structure in the process interface is replaced by new channel and channel structure names for the components. It depends on the type of the constructor C, whether channels or channel structures are introduced for its components.

On the system level, newly created channels and channel structures replace the decomposed channel structure as specified in the following system level transition rule.
Let $a = (\mathbf{split}, (o^{struct}, type),\ C\ o_1 \ldots o_k, \ldots)$ as in the above rule,
$P_{send} = \langle I_s,\ O_s \oplus (o^{struct}, type),\ eqs_{send}\rangle$ with $P_{send} \vdash_a P'_{send}$

$$
\begin{array}{|l|}
\hline
\qquad \mathcal{S} \oplus \ \ \{P_{send},\ \ \langle (o^{struct}, type), (p_s, p_r), \perp\rangle\} \\
\models\ \ \mathcal{S} \oplus \ \ \{P'_{send}, \langle o_1, (p_s, p_r), \perp\rangle \\
\qquad\qquad\qquad\qquad\qquad \ldots, \langle o_k, (p_s, p_r), \perp\rangle\} \\
\text{where } eqs'_{rec} = eqs_{rec} \oplus \{o^{struct} = C\ c_1 \ldots c_k\} \\
\text{and } o_i = (c_i, tg_i) \text{ for } i = 1, \ldots, k \\
\hline
\end{array}
$$

When the receiver then reads the contents of the channel structure o^{struct}, it finds constructor C and the references $o_1 \ldots o_k$ to the component channels. By a straightforward extension of the transitions for receive, the receiver can then introduce $o_1 \ldots o_k$ into its interface and introduce the substitution $o^{struct} = Cc_1 \ldots c_k$ in the environment (see [5]) for details). In this way, input from channel structures is performed uniformly and in a demand-driven way.

3.2 Example evaluations

In order to make clearer the mechanisms described above, we will return to Example 1 and sketch the steps that have to be carried out.

Example 4

The **create**-action for the `child` process has the form shown below. Component p is the identifier of the new process, and the subsequent three components contain its input interface, output interface and local definitions. There is some additional information related to optimizations which will not be considered here.

$$(\mathbf{create},\ \mathrm{p},\ \{in_1\}\ ,\ \{(os_0, [< [\mathtt{Int}] >])\},\ eqs,\ \ldots)$$

In order to build up the outport list with n elements in the child's interface, $n+1$ local steps are carried out, each of them connected with a **split** action:

step	local transition	output interface of child	
0			$\{(os_0, [< [\texttt{Int}] >])\}$
1	$\vdash_{(\mathbf{split},(os_0,[<[\texttt{Int}]>]),c_1:(os_1,[<[\texttt{Int}]>]),\ldots)}$	$\{o_1,$	$(os_1, [< [\texttt{Int}] >])\}$
2	$\vdash_{(\mathbf{split},(os_1,[<[\texttt{Int}]>]),c_2:(os_2,[<[\texttt{Int}]>]),\ldots)}$	$\{o_1, o_2,$	$(os_2, [< [\texttt{Int}] >])\}$
	$\vdots$	$\vdots$	$\vdots$
n	$\vdash_{(\mathbf{split},(os_{n-1},[<[\texttt{Int}]>]),c_n:(os_n,[<[\texttt{Int}]>]),\ldots)}$	$\{o_1, o_2, \ldots, o_n,$	$(os_n, [< [\texttt{Int}] >])\}$
n+1	$\vdash_{(\mathbf{split},(os_n,[<[\texttt{Int}]>]),[],\ldots)}$	$\{o_1, o_2, \ldots, o_n\}$	

This structure will be built up upon encountering the first element in the integer list and entering the definition of `bucket`, which creates a nonempty list for the sublist with remainder `r` and closures for the other sublists. Consequently, the actions will be raised nearly at the same time. This fact may offer possibilities for significant optimizations. The table below shows the development of channel structures and channels in the system over time. OS_i represents the channel list $\langle (os_i, [< [\texttt{Int}] >]), (p_{child}, p_{parent}), cont_i \rangle$ and C_j the channel $\langle o_j, (p_{child}, p_{parent}), cont_j \rangle$:

step	channel structures in the system						channels in the system
0	OS_0						
1	$\vdots$	OS_1					C_1
2	$\vdots$	$\vdots$	OS_2				$C_1 \oplus C_2$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$			
$n-1$	$\vdots$	$\vdots$	$\vdots$		OS_{n-1}		$C_1 \oplus C_2 \oplus \ldots \oplus C_{n-1}$
n	$\vdots$	$\vdots$	$\vdots$		$\vdots$	OS_n	$C_1 \oplus C_2 \oplus \ldots \oplus C_{n-1} \oplus C_n$
$n+1$	OS_0	OS_1	OS_2	$\ldots$	OS_{n-1}	OS_n	$C_1 \oplus C_2 \oplus \ldots \oplus C_{n-1} \oplus C_n$

The development of the receiver mimicks that of the sender, provided that the receiver accesses the contents of the channel structure. It consumes the 'triangle' of channel structures from top to bottom.

◁

We will now reconsider Example 3 and show how the *tree* of channels is built up.

Example 5

The *normalization of the interface* of process abstraction `searchEngine` leads to the following type: `(Tree inf) ->` **Process** `<[req]> Tree <[ans]>`. Thus, the initial interface will consist of one regular (stream) channel for input and a channel structure of type `Tree` of streams for output.

This means that the **create**-action for a `searchEngine` process has a form similar to that of the previous example:

$$(\mathbf{create},\ \mathrm{p},\ \{in_0\}\ ,\ \{(os_0, \texttt{Tree} < [\texttt{ans}] >)\},\ eqs,\ \ldots)$$

The building up of the outport tree with n elements in the child's interface requires n local steps, each of which is connected with a **split** action:

local transition	output interface of child
	$\{(os_0, \texttt{Tree} < [\texttt{ans}] >)\}$
$\vdash$(**split**,(os_0,Tree<[Int]>), Node c_0 (os_1,Tree<[Int]>)(os_2,Tree<[Int]>),...)	$\{c_0, (os_1, \texttt{Tree} < [\texttt{Int}] >), (os_2, \texttt{Tree} < [\texttt{Int}] >)\}$
$\vdots$	$\vdots$
$\vdash$(**split**,(os_m,Tree<[Int]>),Leaf c_{n-1},...)	$\{c_0, c_1, \ldots, c_{n-1}\}$

The **split**-actions generated above cause modifications in the remainder of the system, particularly in the receiver. Channels and channel structures are independent entities in the system. As a reaction to the **split**-actions shown above, the initial channel tree $\langle (os_0, \texttt{Tree} < [\texttt{Int}] >), (p_{child}, p_{parent}) \rangle$ develops to the set $\{C_0, \ldots, C_{n-1}\}$, where C_j represents the channel $\langle c_j, (p_{child}, p_{parent}), cont_j \rangle$ and $c_j = (ci_j, \mathbf{ch})$ $\triangleleft$

4 Complexity and implementation issues

4.1 Implementation in DREAM

In the following, we will sketch DREAM[3], the abstract machine underlying Eden's parallel implementation and how we can extend it in order to support channel structures.

How DREAM works: See below how the state of one process in the abstract machine DREAM can be shown graphically in the form of a table. We give a brief and rather abstract description of the components of DREAM. Afterwards, a concrete representation is given for the rule (C.4), which describes the implementation of the **split**-action.

code	*as*	*rs*	*us*	**out**
other threads				
heap	σ	**ipt**	**b**	

The part above the double line shows the concurrent threads and the part below lists the components shared among all threads of the process. The state of one individual thread is displayed in the topmost line of the diagram. Among these components, the components *code*, *as* (argument stack), *rs* (return stack) and *us* (update stack) correspond directly to the state of the STG[8], which also contains *heap* and global environment σ. The components displayed in **bold face** have been added for the implementation of Eden:

For every thread, there exists a so-called outport specification (**out**), which consists of the identifier of the corresponding channel and the id of the receiver. Correspondingly, on the side of the receiver, there are input specifications, which are stored in a so-called inport table (**ipt**). Inputs can be accessed by all threads of a process and the data already received is stored in the heap. The entries of the input table map channel identifiers to the corresponding heap address and sender id. The current end of available input information in the heap is indicated by a `queueMe`-closure, the access to which leads to the suspension of the thread. The set of threads blocked on a `queueMe`-closure will be released when new information becomes available in the heap. The component **b** is a counter of blocked threads, which is needed for termination detection.

Channels are represented by outports and corresponding inports. The channel buffers are mapped into the heap of the receiver.

How DREAM can be extended: As has been discussed in the context of the operational semantics before, the handling of channel structures requires type information. DREAM uses a function `sendChan` for the transmission of values via channels. This function will be modified to take an additional type argument in the style of the second order λ-calculus, so that we can work with type information explicitly. Note that a conversion into second order λ-calculus is already carried out in the GHC (cf. [9]). However, it has to be noted that in the GHC the second order λ-calculus is used to preserve types through the entire *compiler*, but we need type related information also at *runtime*, namely the information which data constructors have components of which type.
This problem is overcome by introducing a so-called *type table* explained below. It will be accessed by a new primitive function **split** which handles the dynamic replacement of a channel structure by a data structure with channels and/or channel structures as its components.
The function `sendChan` now has the below form[2]. The primitive functions **sendHead** and **sendVal** are defined in DREAM and transmit individual values on stream resp. one-value channels, and **closeStrm** correspondingly closes a stream channel.

```
sendChan <[t]> l = case l of { Nil          -> closeStrm;
                               Cons {y, ys} -> force y ‘seq‘ sendHead y
                                               ‘seq‘ sendChan ys        }
sendChan  <t>  v = force v ‘seq‘ sendVal v
sendChan   t   v = split t v  -- split :: IType -> Val -> ()
```

Given the above extensions, channel structures can be handled by adding to [3] only the following rule **(C.4)**:

[2] As the first argument, the normal form representation of the type of the channel or channel structure is used.

process i (sender)					
$\mathit{Eval}\ (\textbf{split}\ t\vartheta\ v)$	ρ	ε	ε	ε	(j, c)
other_threads$_i$					
h_i [$\rho(v) \mapsto \langle C\{b_1, .., b_n\}\rangle$] $\oplus$ type_table	σ_i	ipt$_i$	b$_i$		

process j (receiver)			
threads$_j$			
h$_j$ [a $\mapsto \langle$`queueMe`$, ts\rangle$]	σ_j	ipt$_j$ [$c \mapsto (a, i)$]	b$_j$

$$\Downarrow$$

process i (sender)					
$\mathit{Eval}\,($`sendChan` $t_1\vartheta\ v_1)\rho_1$	ε	ε	ε	(j, c_1)	
$\vdots$					
$\mathit{Eval}\,($`sendChan` $t_n\vartheta\ v_n)\rho_n$	ε	ε	ε	(j, c_n)	
other_threads$_i$					
$h_i \oplus$ type_table	σ_i	ipt$_i$	b$_i$		

process j (receiver)			
ts			
threads$_j$			
h$_j$ [$a \mapsto \langle C\{a_1, ..., a_n\}\rangle$, $a_1 \mapsto \langle$`queueMe`$, [\,]\rangle$, $\vdots$ $a_n \mapsto \langle$`queueMe`$, [\,]\rangle$]	σ_j	ipt$_j$ [$c \mapsto \perp$ $c_1 \mapsto (a_1, i)$ $\vdots$ $c_n \mapsto (a_n, i)$]	b$_j -$ $\lvert ts\rvert$

The *sender* has as output a channel structure c of type $t\vartheta$, where t denotes a type constructor applied to type variables and ϑ the appropriate instantiation. When it evaluates c to a term $C\ b_1 \ldots b_n$, this channel structure will be split up into the components $c_1 \ldots c_n$, connected by the same n-ary constructor C. The global environment σ provides for type t a reference to a type table in the heap that maps data constructors to lists of component types.

$$\sigma(t) \longmapsto \{\ldots, C \to \{t_1, \ldots, t_n\}, \ldots\}$$

The actual type arguments for the calls to **sendChan** are obtained by applying dynamically[3] the substitution ϑ to the above list elements.
In order to handle subsequent splits, one has also to provide table information for all component types which can be the types of channel structures[4]. As the careful reader will remember, the information about the use of channel structures can only be provided in the type specification of the process abstraction and hence the set of type definitions to be stored in this way can be determined at compile time. The size of the table depends on the number of declarations in a program, i.e. will be constant. In practical programs the table will be very small, because the set of types used as channel structures will be restricted and most of these data structures will be recursive. For simplicity, we will analyze all types that occur as outputs of process abstractions contained in the whole program and collect this information in the initial heap h_0.
Note that the new threads for channels and channel structures look identical. They work with a local environment extended with a variable for the respective output expression:

[3]For many cases an optimized static treatment is feasible, see Section 4.3.

[4]Note that a channel structure can be dynamically split up into instances of channel structures of different types. For a more detailed discussion of this situation, see [5, App. A].

$$\rho_i = \rho[v_i \longmapsto Addr\ b_i], v_i \text{ is a new variable}$$

The *receiver* will react to the changes in the sender by inserting the respective constructor term information at the address of the old inport and by introducing the n new inports. The old inport c will be removed from ipt_j, the receiver's table of active inports. The heap address of c points to the constructor application $C\{a_1, \ldots, a_n\}$, where $a_1 \ldots a_n$ denote new inports. These addresses initially refer to `queueMe`-closures in the heap.
A very important feature of channel structures is that, in this way, the receiver sees no difference between a (finite) structure transmitted over a single regular channel and one transmitted using a channel structure, i.e. referential transparency is not affected by the introduction of channel structures.

4.2 Analysis of complexity

In the previous parts, we have presented the handling of channel structures both in the operational semantics and in the abstract machine DREAM. In summary, the steps to be taken relate in the following way:
A **split**-action corresponds to a call to **split**. In the implementation, the following steps have to be carried out: The sender has to perform the table lookup which includes the instantiation of type variables, and has to spawn new threads. The receiver has to build up new heap closures and modify the entries in the inport table and the thread counter. Furthermore, new channel identifiers have to be generated.
A modification of the input interface corresponds to a modification of the inport table. A modification of the output interface corresponds to the spawning of a new thread. When reasoning about complexity, these modifications deserve some attention because they are already covered in the description of **splitting**, but in the 'simpler' versions occur isolated. The analyses needed for *itype_nf* is related to the creation of the type table.

Based on these considerations, we will now present a detailed comparison of the compile-time and runtime costs of the different approaches. Consider an arbitrary example program that uses a list of n channels. The following table shows the overhead of the three approaches which are most interesting, namely: dynamic **splitting** of channel structures, i.e. alternatives 3 and 4 (cf. Examples 4 and 5), the **emulation** of channel lists by dynamic channels shown in Section 2, and the restriction that requires the **instantiation** of channel lists prior to process creation, i.e. alternative 2. In the last case, it is assumed that some syntactic construct is available for expressing the information about the number of channels in the language.

	splitting	**emulation**	**instantiation**
actions	$n+1$	$2n$	–
interface modif.	$2*(n+1)$	$2n$	–
names	$2n+1$	$n+k$	n
messages	–	n	–
analysis	type table	–	upon creation
other	–	artificial synch.	–

It lists the following information:

- the number of additional[5] **actions** and the number of **modifications in the interface**: The difference between splitting and emulation results from the fact that the splitting of a channel structure can be handled by one action, whereas the introduction of one dynamic channel requires both a **generate**- and a **connect**-action. It has however to be taken into account that a **generate/connect** action is cheaper than a **split**-action (cf. the handling of dynamic reply channels in [3]).

  Interface modifications separate from process creation are expected to imply a synchronization overhead for the respective processes that will outweigh additional costs for analyses connected to process creation (see below).

- the number of new **names** needed: splitting introduces channel structures $OS_0 \ldots OS_n$ plus channels $C_1 \ldots C_n$. The emulation needs n dynamic channels and some constant number k of auxiliary channels in the initial interface (2 in the example).

- the number of additional **messages** to be exchanged: the n messages in the case of the emulation are elements of a stream which can possibly be packed into much less than n messages in the implementation, i.e. the information given here is for the worst case.

- the additional **analysis** steps required and **other** costs imposed by the different approaches:

  For dynamic splitting of channel structures, we have to construct and access the *type table*. The computational cost of this handling has been discussed in Section 4.1.

  For the emulation, a further runtime penalty will result from the artificial synchronization caused by phases 1 to 3 (see page 4).

Naturally, this comparison presents alternative 2 in a favourable light. We have shown that this restriction is not satisfactory in general. Nevertheless it may be useful to extend the solution advocated in this paper (alternative 3 resp. 4) by an optimization for recognizing this case.
In general, we suppose that alternative 1 would not be much more efficient than alternative 2, because even constant patterns will not be evaluated automatically because of lazy evaluation. On the other hand, this approach has to be kept in mind when designing compile-time analyses.

4.3 Optimizations

In order to curb the costs for handling channel structures, especially in the most frequent cases, the following situations should be recognized by specific optimizations:

- As of now, the handling of channel structures is dynamic *only*. The comparison in Section 4.2 indicates that this means unnecessary complexity in cases where the structure is known statically. This is connected to direct implementation support for skeletons[7], which is an important area for future work.

- For lists of channels, considerable savings in the generation of intermediate channel structures would be possible, if multiple channels could be introduced at the same time. In practice, the case that larger parts of a list structure become available (nearly) simultaneously, may be frequent.

[5] i.e. without counting the **create**-action

- Example 2 shows a case where starting `n` different threads in the child process would not pay off, because all of them would be fed by the evaluation of the same expression `bucket n inp`. The first thread that enters the updatable closure for the result expression will probably evaluate the expression in full and leave all other threads blocked on `queueMe` closures. The exact behaviour depends on thread scheduling in the sender process. Here, for the receiver the existence of `n` separate channels is vital. In the sender, the spawning of `n` threads would however be rather counterproductive and should therefore be suppressed.
- In this section, we have described the access to the component types of channel structures as dynamical. It can be done statically for a quite wide range of applications. When only unary type constructors are involved, one can e.g. work with a limited[6] abstract domain.

5 Related work

There is no closely related work in the literature, because other concurrent functional languages do not use explicit channels in the way Eden does.
The concurrent functional language **Erlang**[2] supports the dynamic creation of processes by a built-in function `spawn` which returns the unique process identifier of the newly created process. This identifier can be used by the generator process to send messages to the new process. As messages can contain process identifiers, bidirectional and more general commmunication schemes can be established.
In Concurrent Clean [10, 11] processes correspond to annotated expressions. Communication is completely implicit and implemented by lazy graph copying. Thus, arbitrary information can be transferred between remote processes. But in general it is difficult to figure out the communication topology underlying a parallel process system. The same holds for other functional languages with implicit parallelism.
In the imperative world, the language **SR** offers arrays of 'operations', which can be used like channels[1, Chapter 11.7]. These boundaries can even be input at runtime, which would correspond to alternative 2 in this paper.

6 Further work and conclusions

In this paper we have analyzed different approaches to the use of channel lists and channel structures. We have shown that the dynamic splitting of general channel structures forms an appropriate language feature for Eden.
We have shown that channel structures can be implemented by straightforwardly extending the abstract machine DREAM[3]. Such a powerful implementation will be important for prototyping and skeleton programs, as it provides programs with added generality and flexibility.
What is more, channel structures remove artificial synchronisation constraints: e.g. when the structures to be consumed by one process are already available in full, there will be

[6] Formally there is no limit to the nesting of channel structures and there will have to be one abstract value for every level. Consequently, the size of the domain would depend on the maximal depth of channel structures used in the program. A practical compiler could impose a limit on this.

no need to wait for a complete substructure to become available. In this way, channel structures fruitfully interact with lazy evaluation.
Nevertheless, the costs of the administration of such structures can not be neglected and many cases encountered in practical programs will be comparatively simple. We have proposed special optimizations which can significantly reduce this complexity. The detailed investigation of appropriate mechanisms remains for future work.

Acknowledgements: The authors thank Yolanda Ortega Mallén, Ricardo Peña, Ulrike Klusik and Steffen Priebe for comments on the topic of channel structures.

References

[1] G. R. Andrews and R. A. Olsson. *The SR Programming Language*. Benjamin Cummings, Redwood City, Calif., 1991.

[2] J. Armstrong, M. Williams, and S. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[3] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM - the DistRibuted Eden Abstract Machine. In C. Clack, T. Davie, and K. Hammond, editors, *Symposium on the Implementation of Funct. Lang., St. Andrews*, 1997.

[4] S. Breitinger and R. Loogen. Integration of Channel Structures into the Parallel Functional Language Eden. Internal Report, 1997.

[5] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden — Language Definition and Operational Semantics. Technical Report 96-10, Philipps-Universität Marburg, 1996.

[6] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997.

[7] L. A. Galán, C. Pareja, and R. Peña. Functional Skeletons Generate Process Topologies in Eden. In *International Symposium on Programming Languages: Implementations, Logics, Programs (PLILP)*, LNCS 1140. Springer, 1996.

[8] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *J. of Funct. Prog.*, 2(2), 1992.

[9] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *JFIT*, 93.

[10] M. van Eekelen and R. Plasmeijer. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[11] M. C. J. D. van Eekelen, E. G. J. M. H. Nocker, M. J. Plasmeijer, and J. E. W. Smetsers. Concurrent Clean. Technical Report 89-18, University of Nijmegen, 1989.